

UNIVERSITAT POLITÈCNICA DE CATALUNYA
DEPARTAMENT DE LENGUATGES I SISTEMES INFORMÀTICS
MASTER IN COMPUTING

MASTER THESIS

SPACE-OPTIMIZED TEXTURE ATLASES

STUDENT: JONÀS MARTÍNEZ BAYONA
DIRECTOR: CARLOS ANDÚJAR GRAN
DATE: 8/09/2009

Acknowledgments

We would like to thank TeleAtlas for providing the tridimensional model and facade textures of Barcelona and the Institut Cartogràfic de Catalunya for the aerial photos.

Contents

I	Abstract	7
II	Introduction	8
III	State of the art	10
1	Image comparison metrics	10
1.1	Spatial domain metrics	10
1.1.1	Mean squared error	10
1.1.2	Normalized mean squared error	11
1.1.3	Template matching	11
1.2	Spatial-frequency domain metrics	12
1.2.1	Contrast sensitivity function	12
1.2.2	Mannos-Sakrison's filter	12
1.2.3	Daly's filter	13
1.2.4	Ahumada's filter	13
1.3	Perceptually-based metrics	13
1.3.1	The human visual system	14
1.3.2	Visible differences predictor	14
1.4	Tone mapping metrics	15
1.4.1	Single scale tone reproduction operators	15
1.4.2	Multi scale tone reproduction operators	16
2	Image compression	17
2.1	Lossy compression methods	17
2.2	Lossless compression methods	18
2.3	Compression of images with repetitive patterns	19
3	Texture atlases	20
4	Terrain and urban visualization	22
4.1	Geometry LOD techniques	22
4.2	Texture LOD techniques	22
5	Texture atlas packing	24
5.1	Two-dimensional models	24
5.2	Approximation algorithms	25
5.2.1	Strip packing	25
5.2.2	Bin packing	25
5.2.3	Metaheuristics	25
5.3	Exact algorithms	26
IV	Space-optimized texture atlases	27
6	Overview of the technique	27
6.1	Creation of texture atlases	27
6.2	Real-time visualization	28

7	Creating optimized texture atlases	29
7.1	Image downsampling to match viewing conditions	29
7.2	Image downsampling to match image saliency	30
7.2.1	Generic image metric texture compression	30
7.2.2	Mean squared error metric compression	32
7.2.3	Human visual system metric compression	32
7.3	Texture atlas packing	35
7.3.1	Predicting the minimum size of texture atlas	35
7.3.2	Texture atlas binary tree	35
7.3.3	Optimizing texture space	36
8	Rendering optimized texture atlases	38
8.1	Texture wrapping	38
8.1.1	Mapping of the texture coordinates	38
8.1.2	Compression of the texture coordinates	40
8.1.3	Decompression of the texture coordinates	40
8.1.4	Texture mipmapping and filtering	40
8.1.5	Texture atlas filtering	42
8.1.6	Texture atlas support for DXTC compression	43
8.2	Real-time visualization	45
8.2.1	Texture atlas tree	45
8.2.2	Building rendering	47
8.2.3	Terrain visualization	47
V	Results	49
9	Test specifications	49
9.1	Test model	49
9.2	Hardware tested	50
10	Space compression	51
10.1	Image downsampling	52
10.1.1	Downsampling function of test images	52
10.1.2	Reconstruction of test images with varying RMSE visual tolerance	56
10.1.3	Reconstruction of test images with varying HVSE visual tolerance	61
10.1.4	Test images compression results	69
10.1.5	Image downsampling results of test model	74
10.2	Packing	77
10.3	Encoding texture chart coordinates	81
11	Time Performance	83
12	Selected snapshots	85
VI	Conclusions	88
VII	Appendix	89

List of Figures

1	Images analyzed with RMSE	11
2	Mannos-Sakrison contrast sensitivity function	13
3	Cross Section of the human Eye (Illustration by Mark Erickson)	14
4	Block diagram of the Visible Differences Predictor. Heavy arrows indicate parallel processing of the spatial frequency and orientation channels.	15
5	S3TC lookup table	18
6	Example of a texture atlas	20
7	TileTree by Lefebvre et al. [45] Left: A torus is textured by a TileTree. Middle: The TileTree positions square texture tiles around the surface using an octree. Right: The tile map holding the set of square tiles.	21
8	Distance-dependent texture selection proposed by Buchholz and Döllner [6]	23
9	The Fekete and Schepers modeling approach	24
10	First-Fit Decreasing Height algorithm	25
11	Texture atlas creation scheme	28
12	Example of search space reduction using binary search. Each point of the square represents a texture size (w, h) . <i>Upper row</i> : search on w (first three steps) followed by search on h (three more steps). <i>Lower row</i> : alternating search on w, h . Note that in general both approaches are not guaranteed to find the same (w_o, h_o)	31
13	RMSE error of the subsampling of a facade detail	32
14	Framework of the <i>CIELAB</i> colour model	33
15	HVSE error of the subsampling of a facade detail	34
16	Binary tree structure	36
17	Packed texture coordinates on the atlas	38
18	Encoding of a compressed texture coordinate	40
19	Uninitialized texels at the 2x2 and 1x1 mipmaps for an atlas containing 8x8 and 4x4 textures	41
20	Bilinear filtering of lower mip-levels accesses texels from unrelated neighbouring textures	42
21	Correct bilinear filtering scheme in a 16x16 chart with 2 border texels and 2 mip-levels	43
22	Incorrect bilinear filtering scheme in a 16x16 chart with 1 border texel and 2 mip-levels	43
23	Hierarchical texture atlas representation	45
24	Screen projection factor scheme	47
25	Terrain rendering LOD example	48
26	Thumbnails of a set of Teleatlas textures	49
27	Window downsampling function	52
28	Rocktile downsampling function	52
29	Bricktile downsampling function	53
30	Fabrictile downsampling function	53
31	Textureatlas downsampling function	54
32	Crayons downsampling function	54
33	Boat downsampling function	55
34	Aerial photo downsampling function	55
35	Window reconstruction using RMSE	56
36	Rocktile reconstruction using RMSE	57
37	Bricktile reconstruction using RMSE	57
38	Fabrictile reconstruction using RMSE	58
39	Textureatlas reconstruction using RMSE	58
40	Crayons reconstruction using RMSE	59
41	Boat reconstruction using RMSE	59
42	Aerialphoto reconstruction using RMSE	60
43	Window reconstruction using HVSE	61

44	Rocktile reconstruction using HVSE	62
45	Bricktile reconstruction using HVSE	63
46	Fabritile reconstruction using HVSE	64
47	Textureatlas reconstruction using HVSE	65
48	Crayons reconstruction using HVSE	66
49	Boat reconstruction using HVSE	67
50	Aerialphoto reconstruction using HVSE	68
51	Texture set 1 packing	77
52	Texture set 2 packing	77
53	Texture set 3 packing	78
54	Texture set 4 packing	78
55	Texture set 5 packing	79
56	Texture packing results	79
57	Chart encoding performance (vertices/s)	81
58	Chart encoding performance (fragments/s)	82
59	Chart encoding performance (frames/s)	82
60	Resulting framerate for each technique (walkthrough)	83
61	Framerate evolution of walkthrough	84
62	Barcelona snapshot 1	85
63	Barcelona snapshot 2	85
64	Barcelona snapshot 3	86
65	Barcelona snapshot 4	86
66	Barcelona snapshot 5	87
67	Barcelona snapshot 6	87
68	Some characteristic colours of Barcelona facades	91

List of Tables

1	Space and processing overheads of the three options considered for tiling periodic images	39
2	Area precision and mip-level associated to each atlas tree level of our implementation	46
3	Test model geometry information	49
4	Test model texture information	49
5	Hardware specifications	50
6	Test image specifications	51
7	Texture compression with the whole city	74
8	City downsampling RMSE 5%	75
9	City downsampling RMSE 10%	75
10	City downsampling RMSE 20%	75
11	City downsampling HVSE 10%	75
12	City downsampling HVSE 30%	75
13	City downsampling HVSE 50%	76
14	Texture set packing occupancy	80
15	Chart encoding performance (for more information of the encoding techniques see Table 1)	81
16	Resulting framerate for each technique (walkthrough)	83

List of Algorithms

1	Subsampling image in one direction with error metric	31
2	Compression of an image with error metric	31
3	Texture atlas bin packing	35
4	Inserting an image	36
5	Filling the space between leafs	37
6	Optimizing texture stretch	37
7	Texture wrapping vertex program (see Section 8.1.1)	89
8	Texture wrapping fragment program (see Section 8.1.3)	89
9	Quadtree generation (see Section 8.2.1)	90

Part I

Abstract

Texture atlas parameterization provides an effective way to map a variety of colour and data attributes from 2D texture domains onto polygonal surface meshes. Most of the existing literature focus on how to build seamless texture atlases for continuous photometric detail, but little effort has been devoted to devise efficient techniques for encoding self-repeating, uncontinuous signals such as building facades.

We present a perception-based scheme for generating space-optimized texture atlases specifically designed for intentionally non-bijective parameterizations. Our scheme combines within-chart tiling support with intelligent packing and perceptual measures for assigning texture space in accordance to the amount of information contents of the image and on its saliency. We demonstrate our optimization scheme in the context of real-time navigation through a gigatexel urban model of an European city. Our scheme achieves significant compression ratios and speed-up factors with visually indistinguishable results.

We developed a technique that generates space-optimized texture atlases for the particular encoding of uncontinuous signals projected onto geometry. The scene is partitioned using a texture atlas tree that contains for each node a texture atlas. The leaf nodes of the tree contain scene geometry. The level of detail is controlled by traversing the tree and selecting the appropriate texture atlas for a given viewer position and orientation. In a preprocessing step, textures associated to each texture atlas node of the tree are packed. Textures are resized according to a given user-defined texel size and the size of the geometry that are projected onto. We also use perceptual measures to assign texture space in accordance to image detail.

We also explore different techniques for supporting texture wrapping of uncontinuous signals, which involved the development of efficient techniques for compressing texture coordinates via the GPU. Our approach supports texture filtering and DXTC compression without noticeable artifacts.

We have implemented a prototype version of our space-optimized texture atlases technique and used it to render the 3D city model of Barcelona achieving interactive rendering frame rates. The whole model was composed by more than three million triangles and contained more than twenty thousand different textures representing the building facades with an average original resolution of 512×512 pixels per texture. Our scheme achieves up 100:1 compression ratios and speed-up factors of 20 with visually indistinguishable results.

Part II

Introduction

Heavily-textured models involving a large number of periodic textures are often encountered in many computer graphic applications including architectural visualization, urban modelling and virtual earth globes. Factoring repeated content through texture tiling helps to reduce image data by several orders of magnitude. Current available urban city models, for example, make extensive use of periodic textures to encode highly-detailed repeating patterns on building facades. Even though the current trend in urban modeling is to use real photographs, the limitations of current acquisition technology makes this option feasible only for a few singular buildings, and it is obviously not applicable for ancient sites and non-existing cities. As an example, the most detailed 3D model of the city of Barcelona available today, makes use of a library of 23,939 distinct textures, most of them periodic, to represent 93,111 buildings, in addition to 28 singular buildings for which real photographs are used.

Real-time rendering of detailed textured models is still a challenging problem in computer graphics. Rendering acceleration methods for geometry data (such as maximizing the hit rate of the geometry cache [36], reducing pixel overdraw and level-of-detail rendering) are beneficial for finely-tesselated, geometrically-complex models, but not for urban mass models with a few planar polygons and most details stored in texture maps and displacement maps. Image-based method simplify scene parts by replacing them with impostors, at the expense of high storage requirements. View frustum culling and occlusion culling techniques can provide speed-ups of several orders of magnitude, but are mostly effective for indoor scenes, being of little value e.g. for overhead views of outdoor scenes.

The use of large collections of highly-detailed, periodic textures in models representing man-made structures poses several additional problems:

- *Per-corner attributes.* Polygonal meshes representing man-made structures often exhibit predominance of sharp edges over smooth edges, thus requiring per-corner (rather than per-vertex) attribute binding. A corner is a vertex/polygon pair and typical corner attributes are color, normal, and texture coordinates. Per-corner attribute binding requires much more storage (and often memory bandwidth) than per-vertex binding. For example, a triangle mesh with V vertices has $3T \approx 6V$ corners and a quad mesh has $4Q \approx 4V$ corners, i.e. per-corner binding requires storing six (resp. four) times more attributes than per-vertex binding. For example, a triangle mesh with per-corner normals and texture coordinates typically requires $3 \times 32 \times V$ bits for vertex coordinates and $6 \times (2 + 3) \times 32 \times V$ bits for per-corner attributes.
- *Poor hardware support for per corner binding.* Vertex Buffer Objects (VBOs) is one of the most efficient ways for rendering geometry primitives in today's highly-pipelined hardware. VBOs allow vertex array data to be stored in high-performance graphics memory on the server side and promotes efficient data transfer. Unfortunately, VBOs only support per-vertex binding (the specification of OpenGL does not support multi-index vertex arrays). This means that the only way to use VBOs to render primitives with per-corner attributes is to replicate all per-corner attributes, *including vertex coordinates*. On the example above, the storage space for attributes would be $6 \times (2 + 3) \times 32 \times V$ bits. Even in this scenario rendering VBOs is still much more efficient than rendering individual polygons.
- *Texture switching and texture atlases.* Rendering models using a large collection of texture maps suffers from costly texture switching operations. State-sorting can be performed so that all objects with similar textures are drawn together, but the remaining number of texture switches can be still too high. The traditional approach to avoid texture switching is to pack multiple textures into a single texture atlas. Unfortunately, OpenGL's `GL_REPEAT` wrapping mode cannot be used within a chart inside a texture atlas, and thus tiled textures need to be unfolded on the texture atlas, increasing the required space by several orders of magnitude. Some new GPUs provide a new feature called texture arrays, which allow to reduce texture switching but

all the charts are required to have exactly the same size and the maximum number of charts is limited by the maximum texture resolution rather than by the memory available.

Contribution

In this thesis we present an algorithm for creating a space-optimized texture atlas from a heavily-textured polygonal model with per-polygon textures. Novel elements include:

- A new pipeline to generate a space-optimized texture atlas. Our approach has been conceived to be integrated into a texture LOD management system..
- An efficient algorithm for resizing each chart in accordance with the object-space size of the surface the chart is mapped onto, and the perceptual importance of the chart's contents from given viewing conditions.
- A BSP-based algorithm for packing rectangular charts into a single texture atlas which minimizes unused space. In fact, our algorithm always achieve a 100% of atlas coverage.
- Several shader techniques providing within-chart tiling support for periodic textures. Our strategy avoids unfolding periodic textures. Factoring repeated content through texture tiling helps to reduce image data by several orders of magnitude when compared to conventional texture atlases.

Part III

State of the art

In the following sections we review previous work related to texture atlas generation and parametrization techniques that provide a background to our packing algorithm. We also introduce some previous work on image comparison metrics and image compression techniques strongly related with our perceptual-driven texture compression. A summary of urban and terrain visualization techniques is also presented.

1 Image comparison metrics

Given two images a and b , the value returned by a metric $M(a, b)$ is called the magnitude of difference (MOD) under M . However, the MODs returned by different metrics are not directly comparable. Individual metrics may measure different properties of the images concerned and operate in different sub-domains. Hualin et al.[80] present a study of image comparison metrics to quantify the magnitude of difference between a visualization of a computer simulation and a photographic image captured from an experiment. Normalization of MODs is thus necessary to make it comparable.

In this section, we describe and discuss several metrics classified into three categories: spatial domain, spatial-frequency and perceptually-based approaches.

1.1 Spatial domain metrics

This group of metrics operate in the spatial domain of images and derive an evaluation by examining some statistical properties of images.

1.1.1 Mean squared error

The mean squared error or MSE of an estimator is one of many ways to quantify the amount by which an estimator differs from the true value of the quantity being estimated. MSE measures the average of the square of the error. The error is the amount by which the estimator differs from the quantity to be estimated. The difference occurs because of randomness or because the estimator doesn't account for information that could produce a more accurate estimate.

In the case of image analysis, we examine the MOD between two images pixel by pixel in the form of the squared error of a pair of pixel intensities, and derive its measurement as:

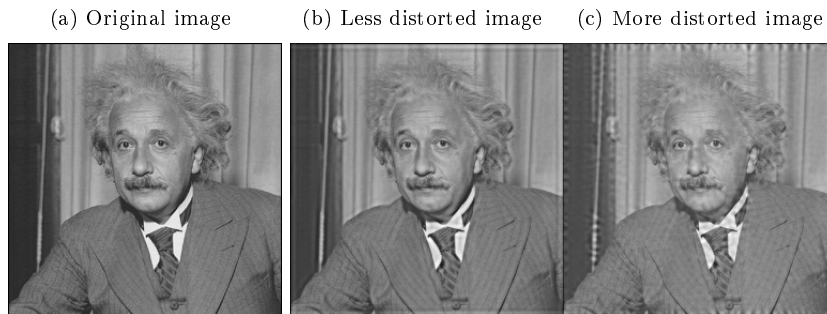
$$MSE = \sum_{x=1}^W \sum_{y=1}^H (a_{x,y} - b_{x,y})^2$$
$$RMSE = \sqrt{MSE}$$

Where a and b are two images of resolution $W \times H$; and $a_{x,y}$ and $b_{x,y}$ are the intensities at pixel (x, y) in a and b respectively. The Root Mean Square Error is just the root of MSE. With colour images the difference between two pixels can be calculated using different colour spaces (RGB, LUV, etc).

MSE is one of the simplest and popular comparison metrics because of their analytical tractability, but it has long been accepted that it is inaccurate in predicting perceived distortion [31].

This is clearly illustrated in the following paradoxical example. Figure 1b and 1c were created by adding different types of distortions to the original image; the original image is shown on Figure 1a. The root mean squared error (RMSE) between each of the distorted images and the original was computed.

Figure 1: Images analyzed with RMSE



The RMSE between the more distorted image and the original is 8.5 while the RMSE between the less distorted image and the original is 9.0. Although the RMSE of the Figure 1b is less than that of Figure 1c, the distortion introduced in the first is more visible than the distortion added to the second. We see that the root mean squared error is a poor indicator of perceptual image fidelity.

Zhou Wang et al. [73] state that MSE is not suitable in the context of measuring the visual perception of image fidelity. The next implicit assumptions are provided in order to demonstrate this poor behaviour:

1. Signal fidelity is independent of temporal or spatial relationships between the samples of the original signal. In other words, if the original and distorted signals are randomly re-ordered in the same way, then the MSE between them will remain unchanged.
2. Signal fidelity is independent of any relationship between the original signal and the error signal. For a given error signal, the MSE remains unchanged, regardless of which original signal it is added to.
3. Signal fidelity is independent of the signs of the error signal samples.
4. All signal samples are equally important to signal fidelity.

Since all image pixels are treated equally in the formulation of the MSE, image content-dependent variations in image fidelity cannot be accounted for.

1.1.2 Normalized mean squared error

Normalized MSE tries to reduce the sensitivity of MSE to the global-shift of image intensities. The compared images a and b are normalized prior to the error calculation. Let μ_a be the mean intensity of image a . The mean of the image is first normalized to 0 by scaling the intensity of each pixel of a as $a'_{x,y} = \frac{a_{x,y}}{\mu_a} - 1$. Let $s_{a'}$ be the standard deviation of the new image a' . The intensity of each pixel of a' is further scaled as $a''_{x,y} = \frac{a'_{x,y}}{s_{a'}}$. The resultant image a'' is thus of a standard deviation 1. the MSE metric is then applied to a'' and b'' , after image b is normalized in the same manner.

1.1.3 Template matching

Template matching is a commonly used technique in pattern recognition [33]. The basic method uses a convolution mask (template), tailored to a specific feature of the search image, which we want to detect. The convolution output will be highest at places where the image structure matches the mask structure, where large image values get multiplied by large mask values.

In order to compare two images we examine the cross-correlation (or auto-correlation) sequences in order to determine if a testing image contains a template image. Consider two images without any object shift. The conventional cross-correlation function of two images a and b is:

$$\gamma(a, b) = \sum_{x=1}^W \sum_{y=1}^H a_{x,y} b_{x,y}$$

However γ is sensitive to the change of luminance of a and b . A better function is:

$$\gamma(a, b) = \sum_{x,y} \frac{(a_{x,y} - \mu_a)(b_{x,y} - \mu_b)}{\sqrt{\sum_{x,y} (a_{x,y} - \mu_a)^2 \sum_{x,y} (b_{x,y} - \mu_b)^2}}$$

Where μ_a and μ_b are the mean intensities of a and b , respectively. Then the MOD metric is defined as:

$$TM(a, b) = \gamma(a, a) - \gamma(a, b) = 1 - \gamma(a, b)$$

Hualin et al.[80] demonstrate that template matching is one of the most effective ways to separate similar and different image groups only surpassed by some fourier comparison metrics (see Section 1.2) and visual differences predictor (see Section 1.3).

1.2 Spatial-frequency domain metrics

Many traditional image comparison metrics operate entirely in the spatial-frequency domain. Images are first normalized and transformed to the Fourier Domain using FFT. A contrast sensitive function (CSF), which models the sensitivity to spatial frequencies, is then applied to the resultant magnitudes. The MOD between the two resultant images is then measured using MSE. In this section, we first introduce CSF and examine different CSF filters in the literature.

1.2.1 Contrast sensitivity function

The contrast sensitivity function (CSF) tells us how sensitive we are to the various frequencies of visual stimuli. If the frequency of visual stimuli is too high we will not be able to recognize the stimuli pattern any more. To apply a CSF we need to know the dimensions of the screen and the distance from we see it.

Imagine an image consisting of vertical black and white stripes. If the stripes are very thin we will be unable to see individual stripes. All what we will see is a grey image. If the stripes then become wider and wider, there is a threshold width from which we are able to distinguish the stripes. The reason why we cannot distinguish patterns with high frequencies is the limited number of photoreceptors in our eye. There are several functions proposed to construct the CSF but the most important are Mannos and Sakrison, Daly and Ahumada filters.

1.2.2 Mannos-Sakrison's filter

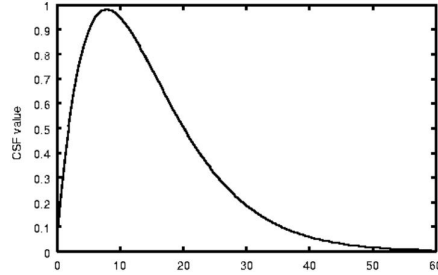
Let f be an image matrix obtained after normalization and FFT, (u, v) be a direction in the Fourier domain (in cycles/deg), and $r = \sqrt{u^2 + v^2}$. Mannos and Sakrison [55] proposed to filter each $f_{u,v}$ using a contrast sensitive function Θ_M :

$$g_{u,v} = f_{u,v} \Theta_M(r)$$

$$\Theta_M(r) = 2.6(0.0192 + 0.144r)e^{-(0.144r)^{1.1}}$$

The spatial frequency of the visual stimuli is defined by r and given in cycles/degree. The function has a peak of value 1 approximately at $f=8.0$ cycles/degree, and is meaningless for frequencies above 60 cycles/degree. Figure 2 shows the contrast sensitivity function Θ_M .

Figure 2: Mannos-Sakrison contrast sensitivity function



1.2.3 Daly’s filter

The Visible Differences Predictor (VDP) proposed by Daly [16] is one of the most well-established algorithms for evaluating image fidelity. It is presented in more depth in Section 1.3.2. Rushmeier et al. [65] adapted the CSF of the VDP in a spatial-frequency domain pipeline for evaluating rendering quality against a captured image. The CSF is applied to $f_{u,v}$ in a way similar to Mannos-Sakrison’s filter, and has the following form:

$$\Theta_D(r) = \left(\frac{0.008}{r^3} + 1 \right)^{-0.2} 1.42r e^{(-0.3r)\sqrt{1+0.06e^{0.3r}}}$$

1.2.4 Ahumada’s filter

Ahumada [40] proposed a CSF that is a balanced difference of two Gaussians as:

$$\Theta_A(r) = a_c e^{-(\frac{r}{f_c})^2} - a_s e^{-(\frac{r}{f_s})^2}$$

where f_c and f_s are the center and surround lowpass cut-off spatial frequency, respectively. a_c and a_s are the center and surround amplitudes. A default set of values may be $a_c = 1$, $a_s = 0.685$, $f_c = 97.32$ and $f_s = 12.16$, but these values can be modified to adjust the sensitivity (see paper for further details). Like the Mannos-Sakrison’s and Daly’s filter, Ahumada’s filter is sensitive to the middle range of spatial-frequencies.

1.3 Perceptually-based metrics

A collection of image comparison metrics, commonly called perceptually-based or HVS metrics, have been developed to simulate some features of the HVS. Many HVS metrics adopt a pipeline which in principle can be viewed as an extension of the general pipeline described in Section 1.2.

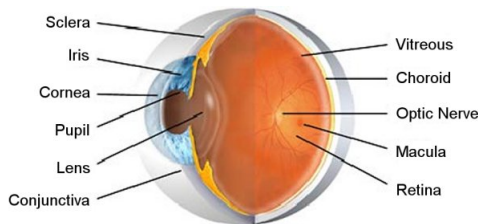
These metrics are of great value for photorealistic rendering. Knowledge of the behaviour of the HVS can be used to speed up rendering by focusing computational effort into areas of an image with perceivable errors. Accounting for such HVS limitations enables computational effort to be shifted from areas of a scene deemed to have a visually insignificant effect on the solutions appearance, and shifted into those areas that are most noticeable. Several attempts have been made to incorporate what is known about the HVS into the image synthesis process.

1.3.1 The human visual system

Perception is the process by which humans, and other organisms, interpret and organize sensation in order to understand their surrounding environment.

The response of the human eye to light is a complex, still not well understood process. It is difficult to quantify due to the high level of interaction between the visual system and complex brain functions. A sketch of the anatomical components of the human eye is shown in Figure 3.

Figure 3: Cross Section of the human Eye (Illustration by Mark Erickson)



The main structures are the iris, lens, pupil, cornea, retina, vitreous humor, optic disk and optic nerve. The path of light through the visual system begins at the pupil, is focused by the lens, then passes onto the retina, which covers the back surface of the eye. The retina is a mesh of photoreceptors, which receive light and pass the stimulus on to the brain.

Some concepts are introduced when talking about Human Visual System:

- *Visual acuity* is the ability of the Human Visual System to resolve detail in an image. The human eye is less sensitive to gradual and sudden changes in brightness in the image plane but has higher sensitivity to intermediate changes. Acuity decreases increasing the distance.
- *Depth perception* is the ability to see the world in three dimensions and to perceive distance. Images projected onto the retina are two dimensional, from these flat images three dimensional worlds are constructed.
- *Perceptual constancy* is a phenomenon which enables the same perception of an object despite it changes in the actual pattern of light falling on the retina.

A number of psychophysical experimental studies have demonstrated many features of how the Human Visual System works. However, problems arise when trying to generalize these results for use in computer graphics. This is because experiments are usually conducted under limited laboratory conditions and are typically designed to explore a single dimension of the HVS.

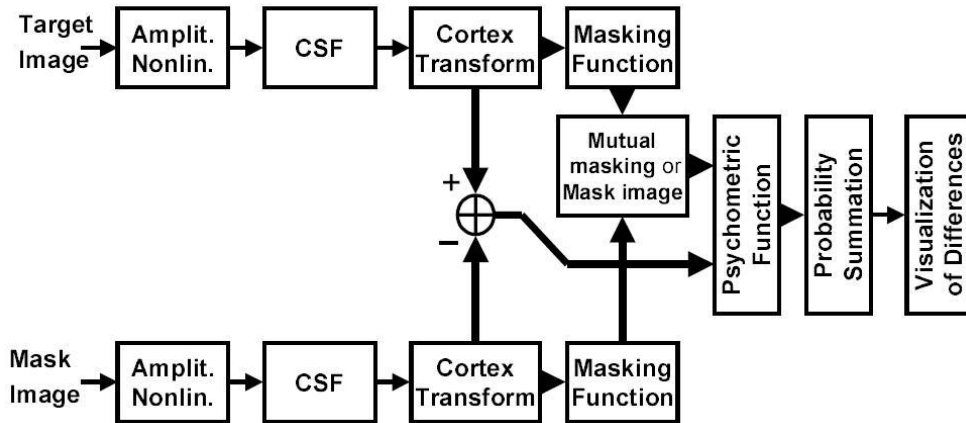
1.3.2 Visible differences predictor

Daly's VDP [16] is a HVS-based image quality metric, which takes two images as input and produces a probability map for difference detection as output. It consists of three main functional components, namely amplitude non-linearity, contrast sensitivity function and detection mechanisms.

- Amplitude non-linearity simulates the adaptation of HVS to local luminance. It applies a non-linear response function to each pixel in the input images, assuming that the adaptation results from an observer fixating a small image area.
- A contrast sensitivity function simulates the variations in visual sensitivity of HVS, and models the variations as a function of spatial frequency. The process is similar to that described in Section 1.2.3, applying a FFT, followed by Daly's CSF, to each image.

- Detection mechanisms simulate the spatial-frequency selectivity of HVS by decomposing each image into 31 independent streams. Multiple detection mechanisms are then applied to the corresponding streams of the two images. These mechanisms include computation of contrasts, application of a masking function to increase the threshold of detectability, and use of a psychometric function to predict the probability of detecting a difference at every location in each stream. Finally, the detection probabilities for all streams are combined into a single image that describes the overall probability for every location.

Figure 4: Block diagram of the Visible Differences Predictor. Heavy arrows indicate parallel processing of the spatial frequency and orientation channels.



1.4 Tone mapping metrics

Tone mapping is a technique used to map a set of colours to another; often to approximate the appearance of high dynamic range images in media with a more limited dynamic range. Print-outs, CRT or LCD monitors, and projectors all have a limited dynamic range which is inadequate to reproduce the full range of light intensities present in natural scenes. Essentially, tone mapping addresses the problem of strong contrast reduction from the scene values to the displayable range while preserving the image details and colour appearance important to appreciate the original scene content.

The human eye is sensitive to relative luminance rather than absolute luminance. Taking advantage of this allows the overall subjective impression of a real environment to be replicated on some display media. Tone reproduction operators can be classified according to the manner in which values are transformed. Single-scale operators proceed by applying the same scaling transformation for each pixel in the image, and that scaling only depends on the current level of adaptation, and not on the real-world luminance. Multi-scale operators take a differing approach and may apply a different scale to each pixel in the image.

Tone reproduction operators are useful for giving a measure of the perceptible difference between two luminance levels at a given level of adaptation. This function can then be used to guide algorithms where there is a need to determine whether some process would be noticeable or not to the end user.

1.4.1 Single scale tone reproduction operators

Tumblin and Rushmeier [71] were the first to apply the dynamics of tone reproduction to realistic image synthesis. Using a psychophysical model of brightness perception first developed by Stevens and Stevens [67] they proposed a tone reproduction operator to match the brightness of the real scene to the brightness of the computed image displayed on a CRT.

Applying Steven’s equation, which relates brightness to target luminance, the perceived value of a real world luminance L_w , is computed as:

$$B_w = 10^{\beta(L_{a(w)})} (\pi \times 10^{-4} L_w)^{\alpha(L_{a(w)})}$$

Where $\alpha(L_{a(w)})$ and $\beta(L_{a(w)})$ are functions of the real world adaptation level:

$$\alpha(L_{a(w)}) = 0.4 \log_{10}(L_{a(w)}) + 1.519$$

$$\beta(L_{a(w)}) = -0.4(\log_{10}(L_{a(w)}))^2 - 0.218 \log_{10}(L_{a(w)}) + 6.1642$$

If it is assumed that a display observer viewing a CRT screen adapts to a luminance, $L_{a(d)}$, the brightness of a displayed luminance value can be similarly expressed:

$$B_d = 10^{\beta(L_{a(d)})} (\pi \times 10^{-4} L_d)^{\alpha(L_{a(d)})}$$

Where $\alpha(L_{a(d)})$ and $\beta(L_{a(d)})$ are as before. To match the brightness of a real world luminance to the brightness of a display luminance, B_w must equal B_d . The luminance required to satisfy this can be determined:

$$L_d = \frac{1}{\pi \times 10^{-4}} 10^{\frac{\beta_{a(w)} - \beta_{a(d)}}{\alpha_{a(d)}}} (\pi \times 10^{-4} L_w)^{\frac{\alpha_{a(w)}}{\alpha_{a(d)}}}$$

This represents the concatenation of the real-world observer and the inverse display observer model. Ward [74] proposed a linear transform with similar result, while reducing computational expense, transforming real world luminances, L_w , to display luminances, L_d , through m , a scaling factor:

$$L_d = mL_w$$

The consequence of adaptation can be thought of as a shift in the absolute difference. The scaling factor, m , dictates how to map luminances from the world to the display such that a Just Noticeable Difference (JND) in world luminances maps to a JND in display luminances. JND is the smallest detectable difference between a starting and secondary level of a particular sensory stimulus (the luminance in the related case).

A critical aspect of tone mapping is the visual model used. As we move through different environments or look from place to place within a single environment our eyes adapt to the prevailing conditions of illumination both globally and within local regions of the visual field. These adaptation processes have dramatic effects on the visibility and appearance of objects and on our visual performance.

1.4.2 Multi scale tone reproduction operators

After an investigation of the effects of tone mapping in a small scene illuminated only by a single incandescent bulb, Chiu et al. [11] realized it was incorrect to apply the same mapping to each pixel. By uniformly applying tone mapping operator across the pixel of an image, incorrect results are likely. They noted that the mapping applied to a pixel should be dependent on the spatial position in the image of that pixel. Using the fact that the human visual system is more sensitive to relative changes in luminance rather than absolute levels, they developed a spatially non-uniform scaling function for high contrast images. First the image is blurred to remove all the high frequencies, and the result is inverted. This approach was capable of reproducing all the detail in the original image, but reverse intensity gradients appeared in the image when very bright and very dark areas were close to each other.

Larson et al [43] developed a histogram equalization technique to create an image with the dynamic range of the original scene compressed into the range available on the display device which was subject to certain restrictions regarding limits of contrast sensitivity of the human eye.

2 Image compression

Texture mapping is employed on rendering systems to increase the visual complexity of a scene without increasing its geometric complexity [61]. Texture compression can help to achieve higher graphics quality with given memory and bandwidth or reduce memory and bandwidth consumption without degrading quality too much. There are many compression techniques for images, most of which are geared towards compression for storage or transmission. In choosing a compression scheme for texture mapping there are several issues to consider:

- *Decoding speed.* in order to render directly from the compressed representation, an essential feature of the compression scheme is fast decompression so that the time necessary to access a single texture pixel is not severely impacted.
- *Random access.* It is difficult to know in advance how a renderer will access a texture. Thus, texture compression schemes must provide fast random access to pixels in the texture.
- *Compression rate and visual quality.* While lossless compression schemes will perfectly preserve a texture, they achieve much lower compression rates than lossy schemes. However, using a lossy compression scheme introduces errors into the textures.
- *Encoding speed.* Texture compression, however, is an asymmetric application of compression, since decoding speed is essential while encoding speed is useful but not necessary.

Image compression can be lossy or lossless. Lossless compression is sometimes preferred for artificial images such as technical drawings, icons or comics. This is because lossy compression methods, especially when used at low bit rates, introduce compression artifacts. Lossless compression methods may also be preferred for high value content, such as medical imagery or image scans made for archival purposes. We classify image compression techniques in the next two sections. Finally we discuss several approaches to compress textures with repeated content.

Texture compression is a specialized form of image compression designed for storing texture maps in 3D computer graphics rendering systems. Unlike conventional image compression algorithms, texture compression algorithms are optimized for random access. Most texture compression algorithms involve some form of fixed-rate lossy vector quantization of small fixed-size blocks of pixels into small fixed-size blocks of coding bits, sometimes with additional extra pre-processing and post-processing steps. Block Truncation Coding is a very simple example of this family of algorithms.

Because their data access patterns are well-defined, texture decompression may be executed on-the-fly during rendering as part of the overall graphics pipeline, reducing overall bandwidth and storage needs throughout the graphics system. As well as texture maps, texture compression may also be used to encode other kinds of rendering map, including bump maps and surface normal maps. Texture compression may also be used together with other forms of map processing such as MIP maps and anisotropic filtering.

2.1 Lossy compression methods

- *Indexed colour.* Indexed colour is the first texture compression algorithm proposed [8] as well as the first to be considered for implementation in hardware [41]. A palettized texture has a colour table, which contains a number of colours stored at high precision. For each texel an index into the colour table is stored. This results in one indirection per texel lookup which is the reason palettized textures are rarely used today. Indexed colour can be considered a special case of vector quantization by treating the individual texels as vectors in colour space.
- *Vector quantization.* Images can be compressed using a codebook of image blocks [30, 4]. Vector quantization uses small blocks (e.g. 4×4 pixels), because with larger blocks the codebook cannot accurately capture the wide variety of block content. By operating on small blocks, vector quantization is effectively exploiting local data correlation.

- *Chroma subsampling.* This technique takes advantage of the fact that the eye perceives spatial changes of brightness more sharply than those of colour, by averaging or dropping some of the chrominance information in the image [10].
- *Block decomposition.* Block decomposition approach is based on the subdivision of an image into small blocks with local colours. S3 Texture Compression (S3TC), sometimes also called DXTn or DXTC, is an implementation of block decomposition technique. It breaks a texture map into 4 x 4 blocks of texels. For opaque texture maps, each of these texels is represented by two bits in a bitmap, for a total of 32 bits. In addition to the bitmap, each block also has two representative 16 bit colours in RGB565 format associated with it. These two explicitly encoded colours, plus two additional colours that are derived by uniformly interpolating the explicitly encoded colours, form a four colour lookup table. This lookup table is used to determine the actual colour at any texel in the block. In total, the 16 texels are encoded using 64 bits, or an average of 4 bits per texel. Decoding blocks compressed in S3TC format is straightforward. A two-bit index is signed to each of the 16 texels. A four colour lookup table (see Figure 5) is then used to determine which 16-bit colour value should be used for each texel. The decoder can be operated at very high speeds and replicated to allow parallel decoding for very high performance solutions.

Figure 5: S3TC lookup table

				color 00	01	01	00	10
				derived				
				color 10	11	00	00	10
				color 11	00	00	10	11
				color 01	00	00	11	01

- *Transform coding.* In transform coding, knowledge of the application is used to choose information to discard, thereby lowering its bandwidth. The remaining information can be compressed then via a variety of methods. When the output is decoded, the result may not be identical to the original input, but is expected to be close enough for the purpose of the application. The common JPEG image format is an example of a transform coding, one that examines small blocks of the image and averages out the colour using a discrete cosine transform to form an image with far fewer colours in total.
- *Fractal compression.* The fractal compression technique [37] relies on the fact that in certain images, parts of the image resemble other parts of the same image. Fractal algorithms convert these parts, or more precisely, geometric shapes into mathematical data called fractal codes which are used to recreate the encoded image. Fractal compression differs from pixel-based compression schemes such as JPEG, GIF and MPEG since no pixels are saved. Once an image has been converted into fractal code its relationship to a specific resolution has been lost; it becomes resolution independent.

2.2 Lossless compression methods

- *Run-length encoding.* Run-length encoding is a very simple form of data compression in which runs of data (that is, sequences in which the same data value occurs in many consecutive data elements) are stored as a single data value and count, rather than as the original run. This is most useful on data that contains many such runs: for example, relatively simple graphic images such as icons, line drawings, and animations. Recently, Carlos Andujar and Jonas Martinez [2] developed a locally-adaptive texture compression scheme encoding homogeneous image regions through arbitrarily-sized texel runs and using a cumulative run-length encoding supporting fast random-access allowing real-time GPU decompression.

- *Entropy encoding.* Entropy coding creates and assigns a unique prefix code to each unique symbol that occurs in the input. These entropy encoders then compress data by replacing each fixed-length input symbol by the corresponding variable-length prefix codeword. The length of each codeword is approximately proportional to the negative logarithm of the probability. Therefore, the most common symbols use the shortest codes.
- *Adaptive dictionary algorithms.* LZW compression [81] is an example of an adaptive dictionary algorithm. A particular LZW compression algorithm takes each input sequence of bits of a given length and creates an entry in a table for that particular bit pattern, consisting of the pattern itself and a shorter code. As input is read, any pattern that has been read before results in the substitution of the shorter code.

2.3 Compression of images with repetitive patterns

Compression of images with repetitive patterns is an interesting problem on the field of image compression. They can be readily observed in man-made environments: buildings, wallpapers, floors, tiles, windows, fabric, pottery and decorative arts; and in nature: the arrangement of atoms, honeycomb, animal fur, gait patterns, feathers, leaves, waves of the ocean, and patterns of sand dunes. To simulate the real world on computers faithfully, textures with repetitive patterns deserve special attention. Humans have an innate ability to perceive and take advantage of symmetry [50]. Rao and Lohse [64] show that regularity plays an important role in human texture perception.

Mathematically speaking, regular texture refers to periodic patterns that present non-trivial translation symmetry, with the possible addition of rotation, reflection and glide-reflection symmetries [57]. When studying periodic patterns, a useful fact from mathematics is the answer to Hilbert’s 18th problem: there is only a finite number of symmetry groups for all possible periodic patterns in dimension n . When $n = 1$ there are seven frieze groups, and when $n = 2$ there are 17 wallpaper groups. Here group is referring to the symmetry group of a periodic pattern. A symmetry group is composed of transformations that keep the pattern invariant.

Most of the work on compressing near-regular images is related to the field of texture synthesis: the process of algorithmically constructing a large digital image from a small digital sample image by taking advantage of its structural content. Procedural texture synthesis [59] is based on generative models of texture. The sample-based approach does not require any previous texture model, yet has the capability of reproducing textures with similar appearance as the input sample [34].

The class of textures that yield good results for texture-synthesis algorithms remains an open question. Lin et al. [51] compare several texture-synthesis algorithms. Their results show that general purpose synthesis algorithms fail to preserve the structural regularity on more than 40% of the tested samples. These results demonstrate that faithful near-regular texture synthesis remains a challenging problem.

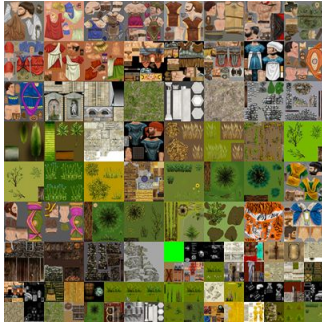
Vector quantization (on page 17) is a good option when compressing near-regular textures, but the difficulty is that even if the image content is highly repetitive, the rigid placement of the blocks implies that they will most often be unique. Other techniques try to make more adaptive the placement of the blocks. Liu et al. [77] and Hays et al. [38] analyze near-regular textures to infer lattice structures and local deformations.

Epitomic analysis of the image is also used [39]. The epitome of an image is its miniature, condensed version containing the essence of the textural and shape properties of the image and is built from the patches of various sizes from the input image. Wang et al. [72] propose an epitomic analysis that enables random-access reconstruction of the image, making it more suitable for interactive applications.

3 Texture atlases

The lack of batching decreases the performance of graphical applications. A batch consists of a number of render-state changes followed by a draw-call. Submitting hundreds or worse, thousands of batches per frame inevitably makes an application CPU-limited due to inherent driver overhead.

Figure 6: Example of a texture atlas



The use of texture atlases reduces the number of batches caused by having to repeatedly bind different textures. The atlas comprises a set of charts, each of which maps a connected part of the 3D surface (a patch) onto a piece of the 2D texture domain. Models using these packed atlases need to remap their texture coordinates.

One way proposed to generate an atlas is to make a chart for each triangle. The simplified triangles are then packed into texture space and sampled to generate texture maps [66]. However seams may appear between triangles due to bilinear interpolation between adjacently packed triangle charts.

Alternatively, triangles may be clustered into patches which are then parameterized as charts [22]. However if the matching boundary edges differ in length or orientation in the texture domain, it is still difficult to eliminate subtle seams along the boundaries (even if a one texel padding is applied just outside the charts).

Recognizing that seams are an important problem with atlases, various approaches have been developed to minimize their effect. The seams may be forced into regions of high negative curvature [49] making them less apparent. As an alternative, an image fidelity metric [79] can be used to minimize the visual effect of seams.

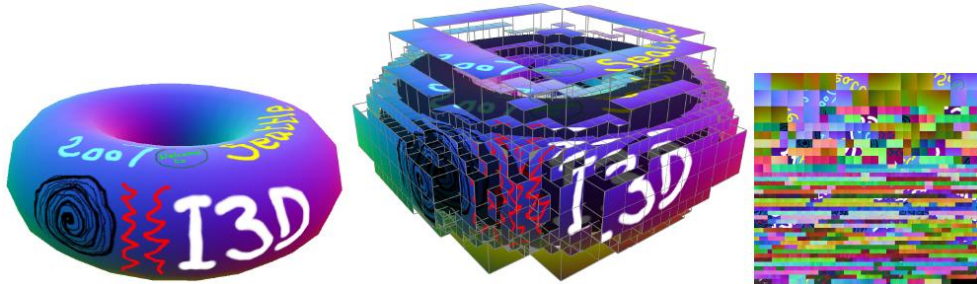
Work on procedural solid texturing has produced a multi-resolution texture atlas [9], which uses standard mip-mapping on graphics hardware. This texture atlas has several desirable properties, including control of the sampling rate across the surface and efficient use of the entire texture space. However, the scheme used still generates seams between charts except at the highest-resolution mip-map level.

Newer approaches avoid seams by parameterizing the surface onto regular charts [62]. While stored discontinuously, neighbouring charts have corresponding samples and a continuous interpolation can be defined along the surface. To avoid splitting the geometry along chart boundaries Tarini et al. [69] parameterize surfaces on the faces of a regular polycube: a set of fixed size cubes surrounding the object. Polycube maps define a continuous, tileable texture space. However the fixed resolution has to be carefully chosen to match the geometric features, the construction requires manual intervention and a triangle mesh is required to encode the parameterization.

To enable texturing of implicit surfaces and avoid explicit parameterization altogether, Benson et al. [5] and DeBry et al. [18] proposed to encode texture data in an octree surrounding the surface. This provides low distortion and adaptive texturing, at the expense of a space and time overhead. Such methods are particularly well suited for interactive painting on 3D objects, where the intrinsic adaptive sampling of the octree structure reduces the waste of memory exhibited by fixed-resolution 2D maps. More recently, Lefebvre et al. [45] (see Figure 7) and Lefohn et al. [47] have proposed GPU

implementation of octree-textures, encoding them in simple 2D or 3D textures, adapted to efficient access by the fragment shader.

Figure 7: TileTree by Lefebvre et al. [45] Left: A torus is textured by a TileTree. Middle: The TileTree positions square texture tiles around the surface using an octree. Right: The tile map holding the set of square tiles.



Most of the existing literature focus on how to build seamless texture atlases for continuous photometric detail, but little effort has been devoted to devise efficient techniques for encoding self-repeating, uncontinuous signals such as building facades. When each polygon of the input model has assigned a different texture, seam artifacts do not occur and more simple strategies for packing the charts into an atlas are possible. In this thesis we focus on this type of models.

4 Terrain and urban visualization

Large terrain and city models may contain millions of triangles, still far too many to render interactively by brute force. Moreover, rendering a uniformly dense triangulation can lead to aliasing artifacts, caused by an unfiltered many-to-one map from samples to pixels, just as in texturing without mipmaps [76]. Thus level-of-detail (LOD) control is necessary to adjust the terrain tessellation as a function of the view parameters.

View-dependent LOD algorithms adaptively refine and coarsen the mesh based on screen-space geometric error, the deviation in pixels between the mesh and the original terrain. Screen-space error combines the effects of viewer distance, surface orientation, and surface geometry. In the next sections we describe several methods to achieve LOD when rendering huge terrain and city models.

4.1 Geometry LOD techniques

Terrain LOD algorithms use a hierarchy of mesh refinement operations to adapt the surface tessellation. Algorithms can be categorized by the structure of these hierarchies as follows:

- *Irregular meshes.* This technique requires the tracking of mesh adjacencies and refinement dependencies but provides the best approximation for a given number of faces. Some hierarchies visually softens the transition between two levels of triangulation [15] while others allow arbitrary connectivities[17].
- *Bin-tree hierarchies.* This uses the recursive bisection of right triangles to simplify memory layout and traversal algorithms [52]. However, these semi-regular meshes still involve random-access memory references and immediate-mode rendering.
- *Bin-tree regions.* Bin-tree regions [48] define coarser-grain refinement operations on regions associated with a bin-tree structure. Precomputed triangulated regions are uploaded to buffers cached in video memory boosting rendering throughput.
- *Tiled blocks.* Tiled blocks partitions the terrain into square patches that are tessellated at different resolutions [35, 12]. The main challenge is to stitch the block boundaries seamlessly.
- *Geometry clipmaps.* Geometry clipmaps [54] caches the terrain in a set of nested regular grids centered about the viewer. These grids represent filtered versions of the terrain at power-of-two resolutions, and are stored as vertex buffers in video memory. As the viewpoint moves, the clipmap levels shift and are incrementally refilled with data. Rather than defining a world-space quadtree, the geometry clipmap define a hierarchy centered about the viewer. The approach has parallels with the LOD treatment of images in texture mapping as it is based on texture clipmaps [68] (see Section 4.2).

4.2 Texture LOD techniques

Huge urban models often require textures that exceeds the main memory capacity. The common method for dealing with large textures requires subdividing a huge texture image into small tiles of sizes directly supportable by typical graphics hardware. This approach provides good paging granularity for the system both from disk to main memory and from main memory to texture memory.

The problem of texture complexity has been addressed in several approaches. Clipmaps of Tanner et al. [68] use dynamic texture representation that efficiently caches textures of arbitrarily large size in a finite amount of physical memory. Cline and Egbert [13] proposed a software approach for large texture handling. At runtime, they determine the appropriate mipmap level for a group of polygons based on the projected screen size of the polygons and the corresponding area in texture space. In another terrain viewing application, Lindstrom et al. [60] use the angle at which textures are viewed to reduce texture requests over using a simple distance metric.

Lefebvre et al. [46] proposed a GPU-based approach for large-scale texture management of arbitrary meshes. The novel idea of their approach is to render the texture coordinates of the visible geometry into texture space to determine the necessary texture tiles for each frame. However the method requires a cost-intensive fragment shader and the geometry has to be rendered multiple times per frame.

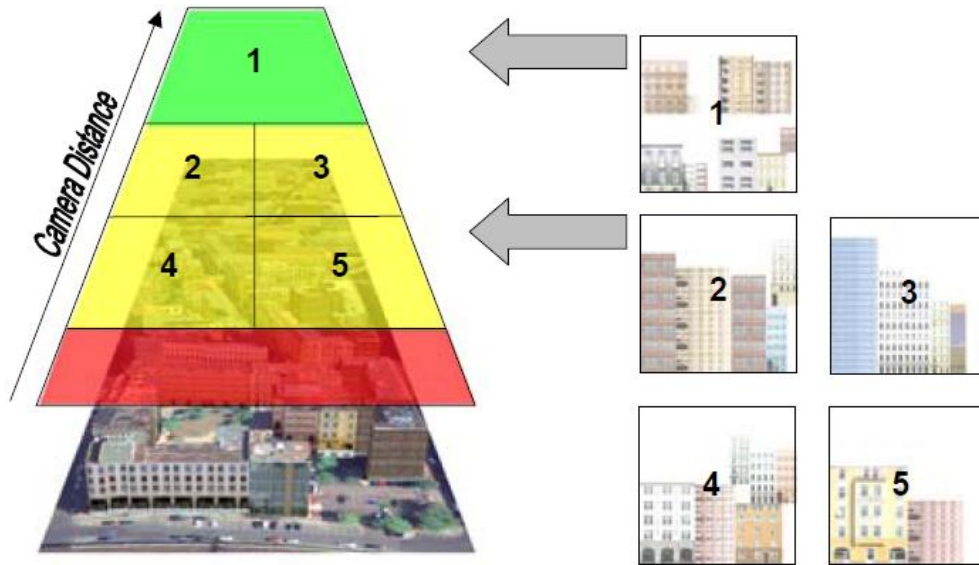
Carr and Hart [9] introduced a texture atlas for real-time procedural solid texturing. They partition the mesh surface into a hierarchy of surface regions that correspond to rectangular sections of the texture atlas. This structure supports mipmapping of the texture atlas because textures of triangles are merged only for contiguous regions on the mesh surface.

Frueh et al. [27] described an approach to create texture maps for 3D city models. The technique includes the creation of a specialized texture atlas for building facades and supports efficient rendering for virtual fly-throughs. Hesina et al. [29] described a texture caching approach for complex textured city models. Their approach is restricted to interactive walk-throughs.

Lakhia [42] proposed an out-of-core rendering engine which applies the cost and benefit approach of the Adaptive Display algorithm by Funkhouser and Sequin [28] to hierarchical levels of detail (HLOD) [23] achieving interactive rendering of detailed city models. To support texturing, they store down-sampled versions of the original scene textures with each HLOD.

Buchholz and Döllner [6] presented a level-of-detail texturing technique that creates a hierarchical data structure for all textures used by scene objects (see Figure 8), and it derives texture atlases at different resolutions. At runtime, their texturing technique requires only a small set of these texture atlases, which represent scene textures in an appropriate size depending on the current camera position and screen resolution.

Figure 8: Distance-dependent texture selection proposed by Buchholz and Döllner [6]



Another way to handle large textures the use of a large virtual texture as a stochastic tiling of a small set of texture image tiles [14]. The tiles may be filled with texture, patterns, or geometry that when assembled create a continuous representation. Li-Yi Wei [75] extended the tile-based texture mapping on graphics hardware.

5 Texture atlas packing

Building a texture atlas from a set of rectangular images involves allocating this set of rectangular images into a larger rectangular image by minimizing the unused space. This is commonly known as the bin packing problem [53]. In two-dimensional bin packing problems these units are finite rectangles, and the objective is to pack all the items into the minimum number of units.

The bin packing is a combinatorial NP-hard problem. The most efficient known algorithms use heuristics to accomplish results which, though very good in most cases, may not be the optimal solution. For example, the first fit algorithm provides a fast but often non optimal solution, involving placing each item into the first bin in which it will fit. It requires $O(n \log(n))$ time, where n is the number of elements to be packed. The algorithm can be made much more effective by first sorting the list of elements into decreasing order, although this does not guarantee an optimal solution, and for longer lists may increase the running time of the algorithm. Most of the contributions in the literature are devoted to the case where the items to be packed have a fixed orientation with respect to the stock unit, one is not allowed to rotate them.

5.1 Two-dimensional models

The first attempt to model two-dimensional packing problems was made by Gilmore and Gomory [58]. They proposed a column generation approach based on the enumeration of all subsets of items (patterns) that can be packed into a single bin. Let A_j be a binary column vector of n elements $a_{ij} (i = 1, \dots, n)$ taking the value 1 if item i belongs to the j th pattern, and the value 0 otherwise. The set of all feasible patterns is then represented by the matrix A , composed by all possible A_j columns ($j = 1, \dots, M$), and the corresponding mathematical model is:

$$\min \sum_{j=1}^M x_j$$

Subject to:

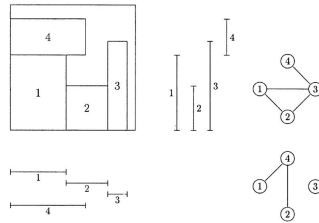
$$\sum_{j=1}^M a_{ij} x_j = 1, \quad (i = 1, \dots, n)$$

$$x_j \in \{0, 1\}, \quad j = (1, \dots, M)$$

Where x_j takes the value 1 if pattern j belongs to the solution, and the value 0 otherwise. Is easy to see the immense number of columns that can appear in A so the only way for handling the model is to dynamically generate columns when needed.

Beasley [3] considered a two dimensional cutting problem in which a profit is associated with each item , and the objective is to pack a maximum profit subset of items into a single bin.

Figure 9: The Fekete and Schepers modeling approach



A completely different modeling approach has been proposed by Fekete and Schepers [26], through a graph-theoretical characterization of the packing of a set of items into a single bin. Let $G_w = (V, E_w)$

be an interval graph with a vertex v_i associated with each item i in the packing and an edge between two vertices (v_i, v_j) if and only if the projections of items i and j on the horizontal axis overlap (see Figure 9). They proved that, if the packing is feasible then:

1. For each stable set S of G_w , $\sum_{v \in S} w_i \leq W$
2. $E_w \cap E_h = \emptyset$

5.2 Approximation algorithms

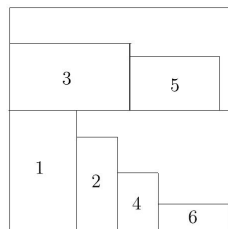
In the next sections we present off-line algorithms (algorithms which have full knowledge of the input). Two classical constructive heuristics algorithms and metaheuristics techniques are presented.

5.2.1 Strip packing

Coffman et al. [21] extended two classical approximation algorithms for one dimension to the two dimensional strip packing problem. They assume that the items are sorted by non-increasing height and packed in levels:

- Next-Fit Decreasing Height (NFDH) algorithm packs the next item, left justified, on the current level (initially, the bottom of the strip), if it fits. Otherwise, the level is closed, a new current level is created (as a horizontal line drawn on the top of the tallest item packed on the current level), and the item is packed, left justified, on it.
- The First-Fit Decreasing Height (FFDH) algorithm packs the next item, left justified, on the first level where it fits, if any. If no level can accommodate it, a new level is created as in NFDH. Figure 10 shows the result of an FFDH packing.

Figure 10: First-Fit Decreasing Height algorithm



5.2.2 Bin packing

Chung et al. [24] studied the following two-phase approach. The first phase of algorithm Hybrid First-Fit (HFF) consists of executing algorithm FFDH of the previous section to obtain a strip packing. Consider now the one dimensional instance obtained by defining one item per level, with size equal to the level height, and bin capacity H . It is clear that any solution to this instance provides a solution to two dimensions. Hence, the second phase of HFF obtains a two dimensional solution by solving the induced one dimensional instance through the First-Fit Decreasing one-dimensional algorithm. The same idea can also be used in conjunction with NFDH and BFDH. The time complexity of the resulting algorithm remains $O(n \log(n))$.

5.2.3 Metaheuristics

Metaheuristic techniques are nowadays a frequently used tool for the approximate solution of hard combinatorial optimization problems. We refer to Aarts and Lenstra [20] for an introduction to this area.

Dowland [19] presented one of the first metaheuristic approaches. His algorithm explores both feasible solutions and solutions in which some of the items overlap. During the search, the objective function is thus the total pairwise overlapping area, and the neighbourhood contains all the solutions corresponding to vertical or horizontal items shifting. As soon as a new feasible solution improving the current one is found, an upper bound is fixed to its height.

5.3 Exact algorithms

An enumerative approach for finding the optimal solution of bin packing was proposed by Martello et al. [56]. Initially, the items are sorted by non increasing height, and a reduction procedure determines the optimal packing of a subset of items in a portion of the strip, thus possibly reducing the instance size.

Fekete and Schepers [25] recently developed an enumerative approach, based on their model (see Section 5.1), to the exact solution of the problem of packing a set of items into a single bin by determining, through binary search, the minimum height such that all the items can be packed into a single bin of base W and height H .

Part IV

Space-optimized texture atlases

In this section we present an efficient technique for generating space-optimized texture atlases for the particular case of 3D buildings. Our method allows the visualization of a huge city in real-time. We also encode self-repeating and uncontinuous signals such as building facades reducing the spatial redundancy and compress these self-repeating details according perceptual measures.

Texture atlases are a well-known technique used to reduce the lack of batching (see Section 3). With the introduction of Texture Arrays it is possible to store a collection of images with identical size and format, arranged in layers. An array texture is accessed as a single unit in a programmable shader, using a single coordinate vector. A single layer is selected, and that layer is then accessed as though it were a one or two-dimensional texture. Some articles used this technique to replace the use of typical texture atlas. Sylvain Lefebvre [44] presents an approach to display properly filtered tilings. Although the benefits of texture arrays are clear (transparent integration, only one binding per array) we still used texture atlas for two main reasons:

1. Texture arrays require all the layered images to have the same dimension. Our approach supports varying texture dimensions to benefit from perceptual-driven and user-defined texel size compression (see Section 7.1 and 7.2).
2. Texture arrays require DirectX 10 or higher disabling the compatibility with several graphical cards.

6 Overview of the technique

Our technique have two main different parts: the creation of the texture atlas and the real-time visualization of the textured buildings. Given a set of textures that are assigned to polygons of the scene, we have to compress and pack them into a single texture atlas. Once we have packed all the required texture atlas, then we are able to visualize the buildings.

6.1 Creation of texture atlases

Our optimization scheme takes as input a tuple (M, \mathcal{I}) , where M is a textured polygonal mesh and $\mathcal{I} = \{I_i\}$ is a collection of color textures. We assume M contains well-defined edges and thus texture coordinates (s, t) are specified per-corner. We also allow input (s, t) coordinates in M to be outside the range $[0, 1]$ to support repeating textures. Without loss of generality we assume $s \geq 0$, $t \geq 0$.

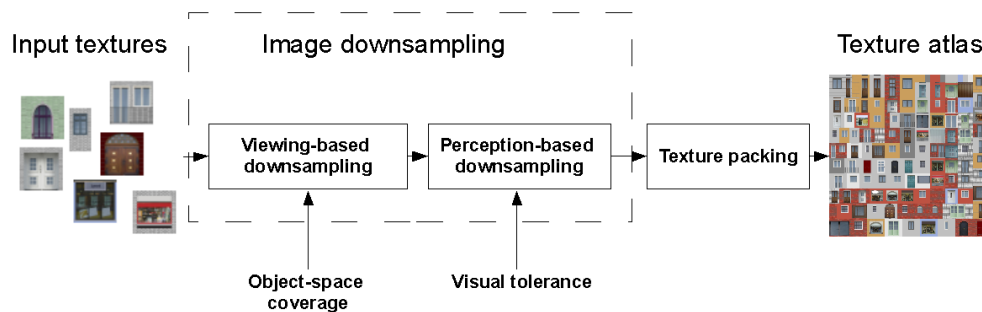
In order to facilitate the integration of our scheme with LOD techniques, the user must provide a parameter l with an upper bound of the desired texel size in object space. This parameter can be easily computed from the viewing range associated with each geometry LOD level so that the screen projection of each texel approximately matches one on-screen pixel.

We introduce in this Section a new pipeline, depicted on Figure 11 to generate a texture atlas:

1. *Computation of object-space texture coverage.* For each texture image, we count the number of times r it is repeated, and the largest surface area $w_s \times h_s$ each tile it is mapped onto. For example, an input image being referenced by a 48×48 quad with texture coordinates $(0, 0)$, $(4, 0)$, $(4, 6)$, $(0, 6)$ has a repeat factor of 24 and each tile maps onto a 12×8 patch of the surface (see Section 7.1).
2. *Image downsampling to match viewing conditions.* Each input image I is downsampled according to the user-defined texel size l (see Section 7.1). For example, the texture atlas for a detailed LOD can be generated using e.g. $l = 5\text{cm}/\text{texel}$, whereas the texture atlas for a coarse LOD might use $l = 50\text{cm}/\text{texel}$.

3. *Image downsampling to match image saliency.* For each image we compute horizontal and vertical stretch factors according to perceptual-based image saliency measures (see Section 7.2). This allows to further reduce the size of some textures with little perceptual impact on the rendered image.
4. *Texture packing.* Texture images are packed into a single texture atlas while minimizing unused space (see Section 7.3.2). We restrict ourselves to rectangular tiles as most tiled textures used in modeling are rectangular to mimic 3D APIs REPEAT wrapping modes.
5. *DXT1 compression [optional].* We also provide support to DXT1 compression minimizing the artifacts and enabling mipmapping (see Section 8.1.5).

Figure 11: Texture atlas creation scheme



6.2 Real-time visualization

As we created all the required texture atlas, we are able to visualize in real-time the whole tridimensional model. Three different parts are described of this process:

- *Texture wrapping.* This stage, which is described in Section 8.1, involves the process of wrapping each chart of the atlas onto the polygons of the scene and achieve a wrapping scheme able to compress efficiently the repetitive patterns. We present a wrapping scheme supporting mipmapping and DXTC compression.
- *Building visualization.* We describe the application fo our atlases to large city visualization. The set of input buildings are partitioned using a quadtree presented in Section 8.2.1. The control of level of detail is done with a texture atlas tree and guarantees a maximum number of texture batches increasing the performance.
- *Terrain visualization.* The terrain is represented with a collection of tiles (see Section 8.2.3). Each tile have associated a set of textures from lowest to highest resolution. The control of level of detail is done in a way similar to the building visualization.

7 Creating optimized texture atlases

7.1 Image downsampling to match viewing conditions

We first downsample the textures according the texture coverage and a user-defined texel size l . Let $w_i \times h_i$ be the dimensions of the input image I_i , and let $w_s \times h_s$ be the surface area each tile of I_i is mapped onto. If $lw_i > w_s$ or $lh_i > h_s$ then we downsample the input image to $(w_s/l, h_s/l)$ using a bilinear filter. Since the resulting downsampled image respects the user-defined texel size in object space, we can safely assume that the image detail lost in this step will be visually indistinguishable under the viewing conditions associated with the LOD level the texture atlas will be used for.

For a set of polygons $P = p_1, p_2, \dots, p_n$ that have associated the same texture we assume that all of them have similar area precision (meters/pixel). We want to obtain a new dimension of the texture that will be inserted in the atlas fitting the new l . In general we have that the texel size coverage lp of a polygon p is:

$$lp_{width}(p) = \frac{P_{width}}{P_{textureWidth}} \quad lp_{height}(p) = \frac{P_{height}}{P_{textureHeight}}$$

And we want the next conditions to match it with the user-defined texel size coverage:

$$l_{width} = \min(lp_{width}(P_i)) \quad l_{height} = \min(lp_{height}(P_i))$$

So the new dimensions for the downsample image I' are calculated as:

$$I'_{width} = \min\left(I_{width}, \max\left(\frac{lp_{width}(P_i)}{l_{width}}\right)\right)$$
$$I'_{height} = \min\left(I_{height}, \max\left(\frac{lp_{height}(P_i)}{l_{height}}\right)\right)$$

Notice that we get the minimum between the downsampled dimension and the available texture space to avoid the use of unnecessary memory space.

7.2 Image downsampling to match image saliency

In Section 1 we reviewed different metrics used to obtain a measure of the similarity between two images. We also have seen that different metrics may measure different properties so it is difficult to compare them. In our case we want to measure the error introduced by the further subsampling of an image. Then, given an image and a maximum error threshold we want to return a new subsampled image with an error not greater than the selected maximum error.

The objective is to construct a perceptual-driven texture compression bounded by a given maximum error. This will let us decrease the image quality within a tolerance. In the next sections we will present a generic texture compression approach and two different image comparison metrics used by our method.

Given an input image I_i , our algorithm must find its smallest downsampled version I_o such that the difference between I_i and its reconstruction \tilde{I}_o from I_o is below an user-defined error threshold, measured by some perceptual-based metric. Therefore our algorithm must search for a pair (w_o, h_o) representing the final size of I_o . We restrict (w_o, h_o) to multiple-of-four values (the motivation of this restriction is explained in detail in Section 8.1.4).

7.2.1 Generic image metric texture compression

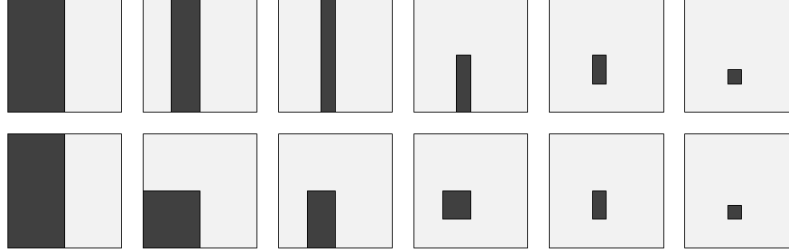
We implemented a texture compression algorithm that reduces the dimensions of an image without surpassing a given error threshold. Let I_i be the input image, I_o the output compressed image, M an image error metric and α the maximum user-defined error. An error value of zero returned by M means that the two images are identical and an error value of one means that they are totally different from the viewpoint of the metric. So we must accomplish the next condition:

$$M(I_i, \tilde{I}_o) \leq \alpha, \alpha \in [0, 1] \quad (1)$$

Since analytical formulae expressing the perceptual image difference in terms of downsampling factor are rather complex, we adopt a much simple approach. Our algorithm performs a search of the optimum (w_o, h_o) values using a dicotomic search. A first option is to first perform a dicotomic search for the minimum width $w_o \in [4..w_i]$ satisfying the error threshold, and then repeat the process for finding the minimum height $h_o \in [4..h_i]$. This algorithm requires $\log_2(w_i) + \log_2(h_i)$ comparisons. For typical 512×512 input size, this amounts for 18 image difference evaluations per input image. However, since (w_o, h_o) are correlated, this approach tends to produce anisotropically-scaled images depending on which coordinate is optimized first. Therefore we adopted a different approach, consisting in searching for the optimum (w_o, h_o) simultaneously, using w_o and h_o alternatively at each step of the binary search (i.e. instead of recursively splitting the 1D interval $[4..w_i]$ and then 1D interval $[4..h_i]$, we split the 2D rectangular interval $[4..w_i] \times [4..h_i]$ alternating horizontal and vertical subdivisions. An example of how the search space for the optimum (w_o, h_o) is reduced at each step of the binary search is shown in Figure 12.

To approximate the dimensions of the output image I_o that satisfies Equation 1, we perform a binary search to detect the minimum size that has an error less or equal than the desired threshold. The algorithm first decreases the image width while the error does not exceed the threshold (see Algorithm 1). Then it decreases the height using the same method. The subsampling of an image is done using a bilinear filter. For better results, the pass is also done decreasing first the image height and then the width (see Algorithm 2). Indeed we have not seen significant differences between the output images when swapping the order of the decrease at each iteration. In order to make it possible the comparison between the subsampled image and the original image, we have to resize the subsampled image to the original size also using a bilinear filter.

Figure 12: Example of search space reduction using binary search. Each point of the square represents a texture size (w, h) . *Upper row*: search on w (first three steps) followed by search on h (three more steps). *Lower row*: alternating search on w, h . Note that in general both approaches are not guaranteed to find the same (w_o, h_o) .



Algorithm 1 Subsampling image in one direction with error metric

```

function subsample_image( $I, \alpha, \text{direction}$ )
 $\alpha_{lower} = 0$  {lower bound}
 $\alpha_{upper} = 1$  {upper bound}
 $\alpha_c = 0$  {current error}
while  $\alpha_c \notin [\alpha - \lambda, \alpha + \lambda]$  do { $\lambda$  is a tolerance threshold}
     $\phi = \frac{(\alpha_{lower} + \alpha_{upper})}{2}$ 
    if decrease direction width then
         $I_o = \text{downsample\_image\_width}(I, \phi)$ 
         $\tilde{I} = \text{restore\_original\_image\_width}(I_o)$ 
    else if decrease direction height then
         $I_o = \text{downsample\_image\_height}(I, \phi)$ 
         $\tilde{I} = \text{restore\_original\_image\_height}(I, \phi)$ 
    end if
     $\alpha_c = M(I, \tilde{I})$ 
    if  $\alpha_c > \alpha$  then
         $\alpha_{lower} = \phi$ 
    else
         $\alpha_{upper} = \phi$ 
    end if
end while
return last valid subsampled image

```

Algorithm 2 Compression of an image with error metric

```

function visual_metric_compression(image, error)
subsampled_wh = subsample_image(image, error, width)
subsampled_hh = subsample_image(subsampled_wh, error, height)
subsampled_hw = subsample_image(image, error, height)
subsampled_hw = subsample_image(subsampled_hw, error, width)
if size(subsampled_wh) > size(subsampled_hw) then
    return subsampled_hw
else
    return subsampled_wh
end if

```

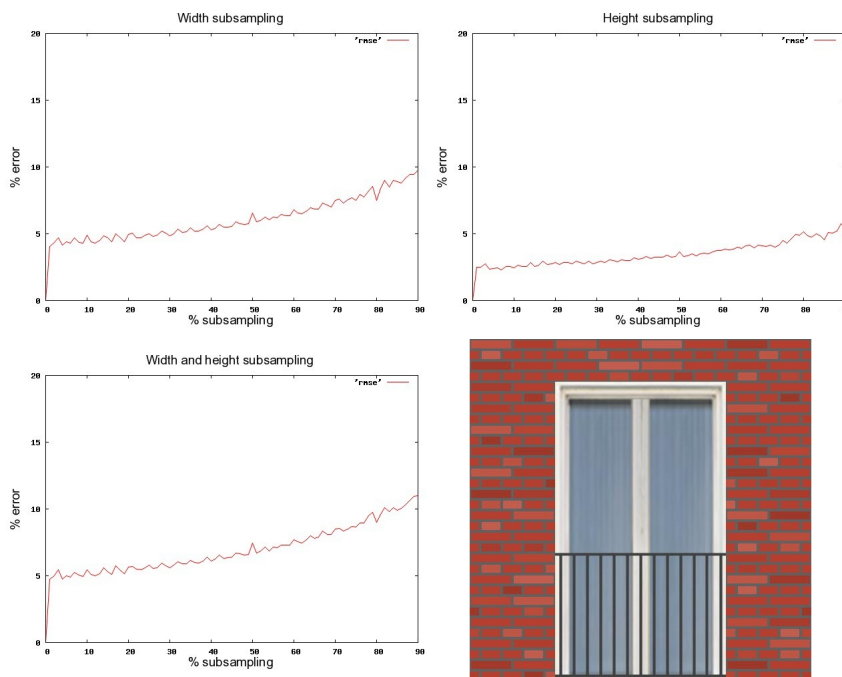
7.2.2 Mean squared error metric compression

The first error metric we tested for the compression is the mean squared error difference that operates on the spatial domain (see Section 1.1.1). For a pair of two pixels we take their intensity and calculate the squared deviation. The global mean squared error is a value between 0 and 1 that quantifies the amount by which the metric estimator differs from the true value of the quantity being estimated. We try to reduce the sensitivity of RMSE to the global-shift of image intensities using the normalized squared error metric (see Section 1.1.2) instead of the typical RMSE.

In Figure 13 we see an estimation of the error caused by subsampling using the RMSE. The original image is a facade detail on the lower right part of the figure. The upper left graphic shows the error due to the width subsampling and the right one because the height. The lower left graphic shows simultaneous subsampling of width and height.

We see that the error tends to increase too much slowly as we increase the subsampling level. This is a well-known problem of RMSE (see Zhou Wang et al. [73]). The difference between one pixel and the corresponding subsampled and resampled pixel of a lower detail level will change slowly; the subsampled colour is interpolated using a bilinear filter and tends to be similar to the pixel with more level of detail. In fact the comparison is done in a locally manner and this is not a desirable property if we are trying to quantify the error of subsampling. So we clearly see that mean squared error does not distinguish well the critical decrease of detail and the perceptual impact that has on the user. More comparisons that illustrates this behaviour are shown in Section 10.1.

Figure 13: RMSE error of the subsampling of a facade detail



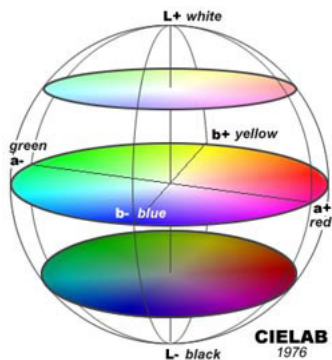
7.2.3 Human visual system metric compression

In order to improve the sensitivity of the RMSE metric we focused on the HVS-based metrics (see Section 1.3).

Our human visual system metric is based on the paper of Yee et al. [78] that describes a perceptually based image comparison process that can be used to tell when images are perceptually identical even though they contain imperceptible differences. Their technique has shown much utility in the

production testing of rendering software and focus on the VDP (see Section 1.3.2). The VDP gives the per-pixel probability of detection given two images to be compared and a set of viewing conditions. Daly’s model takes into account three factors of the Human Visual System (HVS) that reduce the sensitivity to error. The first, amplitude non-linearity, states that the sensitivity of the HVS to contrast changes decreases with increasing light levels. This means that humans are more likely to notice a change of a particular magnitude at low light conditions than the same change in a brighter environment. Secondly, the sensitivity decreases with increasing spatial frequency. For example, a needle is harder to spot in a haystack than on a white piece of paper due to the higher spatial frequency of the haystack. Finally, the last effect, masking, takes into account the variations in sensitivity due to the signal content of the background. Yee et al. [78] also use an abridged version of the VDP in the same way as Ramasubramanian et al. [63] in which they drop the orientation computation when calculating spatial frequencies and extend VDP by including the colour domain in computing the differences.

Figure 14: Framework of the *CIELAB* colour model



We are assuming that the reference image and the image to be compared are in the RGB colour space, so the first step will be the conversion of the images into *XYZ* and *CIELAB* space. *XYZ* is a colour space where *Y* represents the luminance of a pixel and *X*, *Z* are colour coordinates. *CIELAB* is a colour space designed to be perceptually uniform, where the Euclidean distance between two colours corresponds to perceptual distance (see Figure 14). *L* also represents luminance and *A*, *B* are colour coordinates. This conversion step is described in Glassner [32].

To compute the threshold elevation factor *F*, or how much tolerance to error is increased a spatial frequency hierarchy is constructed from the *Y* channel of the reference image. This step is efficiently computed using the Laplacian pyramid of Burt and Adelson [7]. The pyramid enables to compute the spatial frequencies present the image to determine how much sensitivity to contrast changes decreases with increasing frequency. The pyramid is constructed by convolving the luminance *Y* channel with a separable filter.

Following Ramasubramanian et al. [63] we compute the normalized Contrast Sensitivity Function (CSF) multiplied by the masking function given in [16] to obtain the combined threshold elevation factor, *F*. We compute some of the intermediate variables from the field of view (*fov*) and the image width with the following from Ward et al. [43]:

$$pixelsPerDegree = \frac{width \times \pi}{360 \tan\left(\frac{fov}{2}\right)}$$

$$cyclesPerDegree = \frac{pixelsPerDegree}{2}$$

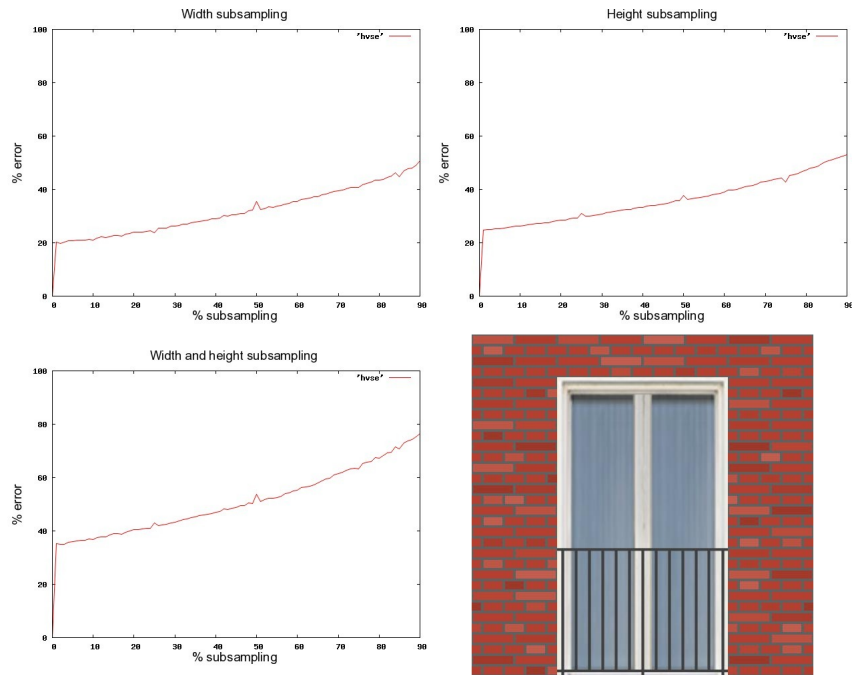
Where *fov* is the horizontal field of view in radians, *width* is the number of pixels across the screen. The top level of the Laplacian pyramid corresponds to frequencies at cycles per degree and each level

thereafter is half the frequency of the preceding level. Typical values for fov and width are discussed in the next section. The CSF is computed as a function of cycles per degree and luminance.

Finally two tests are performed to know if the images are perceptually different. If the difference of luminance between two corresponding pixels in the reference and test images is greater than a threshold, then the first test fails. The threshold is defined by the average of pixels in a one degree radius from the Y channel of the reference image. The second test is performed on the A and B channels of the reference and test images using a scale factor that turns off the colour test in the mesopic and scotopic luminance ranges (nigh time light levels) where colour vision starts to degrade.

We have done some modifications in the original algorithm of Yee et al. [78] to adapt it for our requirements. We do not want to decide only if two images are different or not as we want to quantify this difference. So we take the total number of pixels that failed the HVSE test and divide it by the total number of pixels of the image giving us a value between 0 and 1. Then a value of 0 means that the images are identical and 1 that are totally different in terms of human perception.

Figure 15: HVSE error of the subsampling of a facade detail



In Figure 15 we see an estimation of the error caused by subsampling using the HVSE with the same presentation scheme as used in Figure 13. The luminance for the HVS calculations has been established at a default 100cmd^{-2} . It is important to notice that the y axis of Figure 15 has more error range than the y axis of Figure 13.

As we see HVSE has more sensitivity to the effect of subsampling compared with RMSE in all the three tests and is more reactive to the increase of subsampling. As we have more range of error with HVSE it also means that we have more resolution and furthermore it is designed to react better with the human visual system conditions. So we decided to use HVSE for our compression scheme based on visual tolerance, although it is slower than the classical RMSE. In Section 10.1 we show more examples of this low RMSE sensitivity.

7.3 Texture atlas packing

We need to pack a set of rectangular textures with varying sizes into a single texture atlas. The size of a texture atlas must be a power of two to make an efficient use of texture hardware memory. We use a Binary Space Partition [70] to define the space occupied by each input texture. An overview of our bin packing is shown in Algorithm 3. The input parameters are a collection of textures, an area precision (meters per pixel) and visual metric tolerance.

We first sort the input textures from the biggest to the smallest. This is a common step also used by strip bin packing [21] to optimize the occupied space. The next step, consist in predicting the minimum size of texture atlas and increase it while we do not have enough space. Finally we optimize the size of the inserted textures growing them progressively. In the next sections we will explain in detail each of these mentioned steps.

Algorithm 3 Texture atlas bin packing

```
Sort textures from biggest to smallest
Calculate the minimum size of texture atlas
valid_size = Insert all the textures in the atlas
while (not valid_size) do
    Increase size of atlas
    valid_size = Insert all the textures in the atlas
end while
Optimize inserted textures
```

7.3.1 Predicting the minimum size of texture atlas

We define the initial size of the texture atlas taking in account the sum of all the sizes of input textures. Let $I = i_1, i_2, \dots, i_n$ be the input set of textures. The initial width and heigh are defined as:

$$\alpha = \log_2 \left(\sum_{j=1}^n i_{width_j} \times i_{height_j} \right)$$
$$width = 2^{\lceil \frac{\alpha}{2} \rceil} \quad height = 2^{\alpha - \lceil \frac{\alpha}{2} \rceil}$$

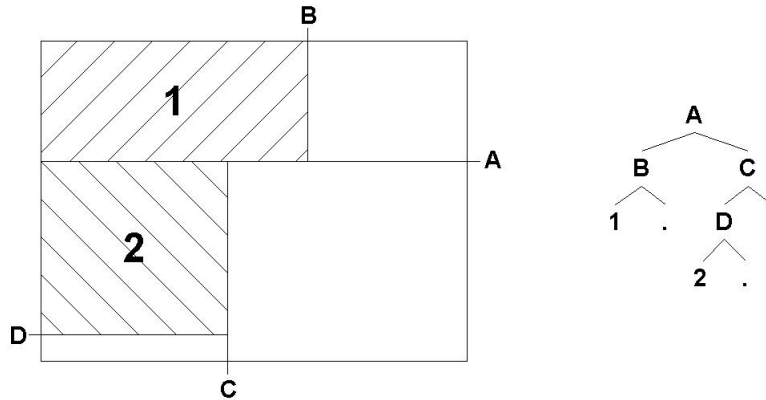
This formula returns a width and height potency of two and minimum size to contain the input textures. To increase the size of the texture atlas to the next greater valid size, we just sum one unit to α and update the corresponding width and height.

7.3.2 Texture atlas binary tree

A node of the texture atlas binary tree defines a rectangular region of the texture atlas. The root of the tree defines the whole texture space. If a node is a leaf, it may be either occupied by a single texture or unoccupied. If not, has two child nodes that overlap all the space of the parent node (see Figure 16).

We call the insert function (see Algorithm 4) passing it the root node and the width and height of the texture atlas. If the texture is too small we just return. We accept if it fits perfectly on the node. Otherwise we split the node in the direction (by width or height) that provides more space for the input image and recursively call the insert function. So the insertion time takes $O(\log(n))$.

Figure 16: Binary tree structure



Algorithm 4 Inserting an image

```

function insert (nod)
if node is empty and is a leaf then
    if is too small then
        return
    end if
    if is just right then
        accept node and return
    end if
    split the node in the direction with more space
    insert (node.left)
else
    insert (node.left)
    insert (node.right)
end if

```

7.3.3 Optimizing texture space

We also optimize the occupied space of input textures after we have inserted them on the hierarchy. Since we have the restriction to only deal with texture sizes power of two, a considerable amount of unused space may appear. The optimization process has two steps explained below.

First we perform a binary search taking the lower bound as the current size of each inserted textures and the upper bound the maximum size of the texture. We just iterate trying to optimize the stretch and obtain the maximum occupied space (see Algorithm 6).

Then for each pair of leafs of the BSP tree, in the case we have one filled with a texture and the other one empty, we just expand the filled one to occupy the empty space (see Algorithm 5). This enables us to obtain a final 100% occupation of the texture atlas. Packing results are described in Section 10.2.

Algorithm 5 Filling the space between leafs

```
function fill_leaf_space (node)
  if node has leafs then
    if right node is empty and left node is filled then
      Expand left node to occupy also right node
    else if left node is empty and right node is filled then
      Expand right node to occupy also left node
    end if
  else
    fill_leaf_space (node.left)
    fill_leaf_space (node.right)
  end if
```

Algorithm 6 Optimizing texture stretch

```
repeat
  for each texture do
    New texture size equals (lower bound size + upper bound size) / 2
  end for
  Insert stretched textures to atlas
  for each texture do
    if valid atlas then
      Update texture lower bound
    else
      Update texture upper bound
    end if
  end for
until not convergence of lower and upper bound
```

8 Rendering optimized texture atlases

8.1 Texture wrapping

Once we have all the textures packed into a single texture atlas we want an efficient method to wrap each self-repeating texture onto the geometry. Furthermore we want to support mipmaps: a pre-calculated, optimized collections of images that accompany a main texture, intended to increase rendering speed and reduce aliasing artifacts [76]. Our method is based on the use of an specialized texture coordinate that compactly represents the repetitive textures. In addition we add a border to each packed texture in order to transparently support mipmapping. In the next subsections we will present first the process of mapping and compression of the original textures, the real-time decompression, the developed way to use mipmapping and finally the requirements to adapt the wrapping using DTXC compression.

8.1.1 Mapping of the texture coordinates

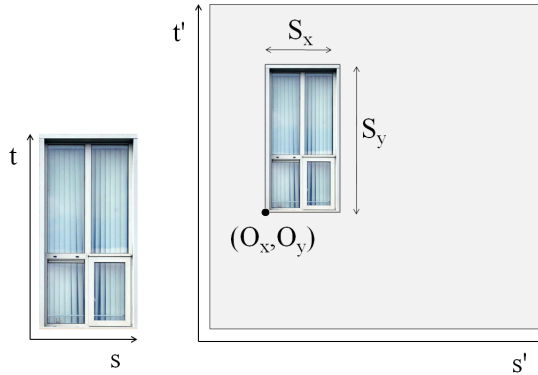
We have tested several ways to encode the texture coordinates of the charts packed into a texture atlas. For any chart we have the next constraints (all the dimensions are in the normalized range $[0, 1]$):

1. Origin of the chart O
2. Size of the chart S

Our texture atlases support periodic texture tiling by mimicking OpenGL's `GL_REPEAT` wrapping mode. Since all the charts are rectangular, this support can be efficiently implemented on a fragment shader with minimum processing overhead. Suppose that, in the input model, a primitive had assigned a periodic texture T . After packing, image T occupies a certain rectangular region of the texture atlas. Let O, S be the origin and the size of T in the texture atlas, in normalized coordinates (see Figure 17). Typically, texture atlas construction implies replacing the local (s, t) coordinates of the primitive by global (s', t') coordinates to reflect the new parameterization. In our texture atlases all charts are axis-aligned rectangles, so conversion to global coordinates can be done using this equation,

$$\begin{bmatrix} s' \\ t' \end{bmatrix} = \begin{bmatrix} S_x & 0 & O_x \\ 0 & S_y & O_y \end{bmatrix} \begin{bmatrix} s \\ t \\ 1 \end{bmatrix}$$

Figure 17: Packed texture coordinates on the atlas



Unfortunately, the above conversion from local to global coordinates is valid only for input (s, t) coordinates in the range $[0, 1]$, which is not the case for tiled textures. Therefore we let the application

	Encoding of (O, S)	Memory Space	App \Rightarrow GPU	Vertex Program	VBO compatibility
Option 1	uniform vec4	4 floats/T	4 floats/T	none	No
Option 2	attribute vec4	4 floats/T	12 floats/T	copy vec4 to varying	Yes
Option 3	As part of s,t coords	none	none	decode s,t,origin,size	Yes

Table 1: Space and processing overheads of the three options considered for tiling periodic images

keep the input coordinates (s, t) in local space. During rasterization, these local coordinates will be interpolated on a per-fragment basis. We then let the fragment shader perform the conversion to global coordinates using this straightforward formula,

$$\begin{bmatrix} s' \\ t' \end{bmatrix} = \begin{bmatrix} S_x & 0 & O_x \\ 0 & S_y & O_y \end{bmatrix} \begin{bmatrix} \text{fract}(s) \\ \text{fract}(t) \\ 1 \end{bmatrix} \quad (2)$$

where now (s, t) are per-fragment interpolated values.

We are multiplying the fractional part of original textures by the chart size and then adding the offset of the origin. The new coordinates (s', t') can be used to query the value on the atlas associated with the original coordinates (s, t) . This local transformation can be easily integrated using the programmable rendering pipeline. We just need to transform in the fragment shader the (s, t) coordinates and use them for the texture lookup.

The algorithm 7 of the appendix shows the simple GLSL shader code of the texture mapping using the mapping function of Equation 2. With the standard texture lookup functions the implicit level of detail is selected as follows: for a texture that is not mip-mapped, the texture is used directly but if it is mip-mapped and running in a fragment shader, the LOD computed by the implementation is used to do the texture lookup. If it is mip-mapped and running on the vertex shader, then the base texture is used. But that is not valid using a texture atlas because we are referring a local coordinate for each chart and we need to specify the derivatives using the local coordinates. So we enable the extension `GL_ARB_shader_texture_lod` in order to use the function `texture2DGrad` that makes a texture lookup with explicit gradients (visit www.opengl.org/registry/doc/GLSLangSpec.Full.1.40.05.pdf for more information).

We have explored three different options for passing the tuple (O, S) to the fragment shader (see table 1). In the following discussion we use GLSL terminology. A first option that suggests itself is to encode (O, S) in a **vec4 uniform** variable that the application must send for each textured primitive. This implies an overhead of storing and transmitting 4 floats per triangle. Unfortunately, the current OpenGL specification does not support binding uniform variables to buffer objects, i.e., uniforms cannot be modified in a vertex array. Therefore this approach is not compatible with using Vertex Buffer Objects (VBOs) for rendering groups of primitives, which is the most efficient rendering mode in current graphics hardware.

A second option is to send (O, S) data as **attributes**. The main advantage is that attributes can be bound to buffer objects and hence the option is compatible with VBOs. The memory requirements for the application are the same as in the previous option. The transmission overhead, though, depends on the rendering mode. For VBOs, the attribute must be specified on a per-corner basis (OpenGL only supports a single index buffer for vertex-arrays). This leads to a transmission overhead of 12 floats per triangle.

The third option we considered attempts to minimize space and transmission overheads (see Section 8.1.2). A key observation is that texture coordinates are represented in 32-bit single precision floating-point format. However, the maximum texture size supported by state-of-the-art graphics hardware is 4096×4096 . Addressing a texel on such a texture requires only 12 bits. Therefore, we can use the unused bits to encode (O, S) data. In our implementation, each (s, t) coordinate is replaced by the

following 32-bit mask.

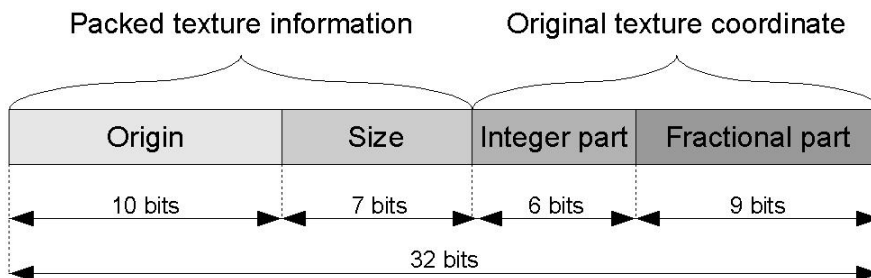
8.1.2 Compression of the texture coordinates

We make some assumptions on the size and position of the charts in order to allow texture coordinate compression. First of all we consider that the maximum size of a texture atlas is 4096x4096 pixels and the maximum size of each chart is of 512x512 pixels. Also the size of the atlas and the charts must be multiple of four so we only need 10 bits to encode the origin and 7 bits to encode the size. About the original (s, t) coordinates, the integer part must be in the $[0..63]$ range (6 bits) and the fractional part in the $[0..511]$ range (9 bits). All these restrictions allows us to encode with two unsigned integers (32 bits) all the required information for texture mapping (see Figure 18):

1. Original texture coordinate s or t (15 bits)
 - (a) Integer part (6 bits)
 - (b) Fractional part (9 bits)
2. The origin of the chart O_{width} or O_{height} (10 bits)
3. The size of the chart S_{width} or S_{height} (7 bits)

So we only have to send two packed components per vertex. We have seen that generally this memory layout fits well with our test application but the amount of bits used for each encoded attribute can be modified to match the requirements of another application.

Figure 18: Encoding of a compressed texture coordinate



8.1.3 Decompression of the texture coordinates

The decompression of the packed coordinates (see Section 8.1.2) is done in the vertex program stage. The two encoded unsigned integers composed by $(s, O_{width}, S_{width})$ and $(t, O_{height}, S_{height})$ are decompressed using the integer operators incorporated in the extension `EXT_gpu_shader4` of Nvidia (visit http://www.opengl.org/registry/specs/EXT/gpu_shader4.txt for more information).

The algorithm 7 of the appendix shows the simple GLSL shader code of the decompression of coordinates.

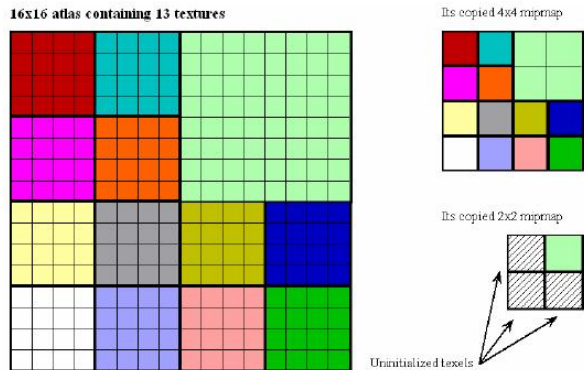
8.1.4 Texture mipmapping and filtering

Mip-mapped textures are essential for achieving any kind of rendering performance. Packing mip-mapped textures into an atlas, however, seems to imply that the mipmaps of these charts combine, until eventually the lowest mip-level of 1×1 resolution smears all textures of an atlas into a single texel. We see that the use of texture mipmapping in combination with texture atlas is a more complex problem than using a single image.

So the first problem arises when we want to determine the lowest mip-level that we are able to use. In the Figure 19 we see a texture atlas with a set of charts represented by different colours. When we

generate the mip-level with a 1 : 8 reduction, we get three texels with an unexpected colour due the incorrect mix of the upper level charts.

Figure 19: Uninitialized texels at the 2x2 and 1x1 mipmaps for an atlas containing 8x8 and 4x4 textures



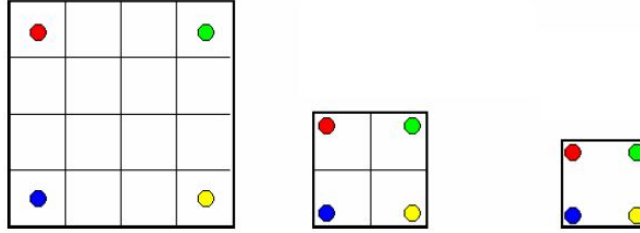
When we pack textures directly into an atlas texels they are never combined (just copied) and no smearing or cross-pollution occurs. But when we generate the mip-map chain, uninitialized texels may appear. In the technical report of Nvidia [1] related to texture atlas, they propose a simple solution for this problem. Even generating mip-map chains of atlases on the fly with a two-by-two box filter does not pollute mip-maps with neighbouring texels, if the atlas is a power-of two texture and contains only power-of-two textures that do not unnecessarily cross power-of-two lines. As the various mip-levels are generated, texels of separated textures do not combine. Also because textures can differ in size and large textures have longer mip-chains than smaller textures, the largest texture packed into an atlas determines the number of mip-map levels in the atlas. So they abridge the mip-chain of an atlas to the length of the mip-chain of the smallest texture contained in the atlas. However, that would typically have severe performances and image quality implications. Furthermore, the restriction that each chart must have a power-of two dimension drastically reduces the available sizes of charts.

More problems overcome when we want to support texture filtering: the method used to determine the texture colour for a texture mapped pixel, using the colours of nearby texels. Artifacts may appear at the borders as we use texels from foreign textures to filter. For the highest resolution mip-level, a possible solution is to clamp to the edge the texture coordinates sampling a texel at its center on the border: even when bilinear filtering (in this method the four nearest texels to the pixel center are sampled at the closest mipmap level, and their colours are combined by weighted average according to distance) is enabled only that texel contributes to the filtered output. So the new texture coordinates must be in the range:

$$(s, t) \in \left(\frac{1}{2width} \dots 1 - \frac{1}{2width}, \frac{1}{2height} \dots 1 - \frac{1}{2height} \right)$$

While bilinear filtering of the highest resolution mip-level is safe, anisotropic filtering of the same mip-level does potentially access unrelated neighbouring texels. Worse, bilinear and anisotropic filtering of all lower mip-maps also access unrelated neighbouring texels, as Figure 20 demonstrates. The most left figure shows the sampling of corner texels at the highest mip-level clamping to the edge, so we do not have neighbour contribution to the filtering. But at the two next figures, corresponding to lower mip-levels, the same coordinates are no longer dead-center, producing an incorrect filtering.

Figure 20: Bilinear filtering of lower mip-levels accesses texels from unrelated neighbouring textures



Taking account of all this mentioned problems, we developed a mipmapping and texture atlas filtering scheme capable to do bilinear filtering and generate a mip-map chain, without the restriction of having only power-of two dimensions of the charts.

The first constraint is introduced by the texture coordinate compression (see Section 8.1.2). Each chart must have a dimension multiple by four. Indeed, this restriction is less hard than to have a power-of two dimension. For example, in a texture atlas with 1024 pixels of width, we have 256 possible dimensions with our method. With the power-of two restriction we have only ten. Indeed, the maximum mip-map levels per atlas is two, as the smallest dimension we may have is four. But that is not a significant problem, as we are using a progressive representation of the level of detail through a hierarchy (see Section 8.2). We use a two-by-two box filter for the mip-map chain generation, guaranteeing that texels of separate textures do not combine until we do not reach more than two mip-levels.

To provide bilinear filtering we thought about two different techniques:

1. The first solution is to use a fragment shader to clamp atlas coordinates to corresponding charts taking into account which mip-level the texture operation is about to access. For the highest mip-level the atlas coordinates remain unchanged, yet for lower mip-levels the atlas coordinates are remapped closer to the center. This technique requires pixel shader 2.0 support and a comparatively complex and expensive shader.
2. The second solution is to pad textures and their mip-chains with border texels. That consumes more space, but provides a transparent integration in the texture filtering pipeline and the mip-map chain does not involve any special requirement aside the typical box subsampling. In the next section we will explain in detail this approach and the integration with our technique.

8.1.5 Texture atlas filtering

We add a border for each chart that replicates the repeating effect. Let b be the size of the border, (O_{width}, O_{height}) the chart origin and (D_{width}, D_{height}) the height. Then we have that the final origin $(Of_{width}, Of_{height})$ and size $(Df_{width}, Df_{height})$ is:

$$(Of_{width}, Of_{height}) = (O_{width} - b, O_{height} - b)$$

$$(Df_{width}, Df_{height}) = (D_{width} + 2b, D_{height} + 2b)$$

The cost of increasing the size with this padding is:

$$Df_{width}Df_{height} = D_{width}D_{height} + 2b(D_{width} + D_{height}) + 4b^2$$

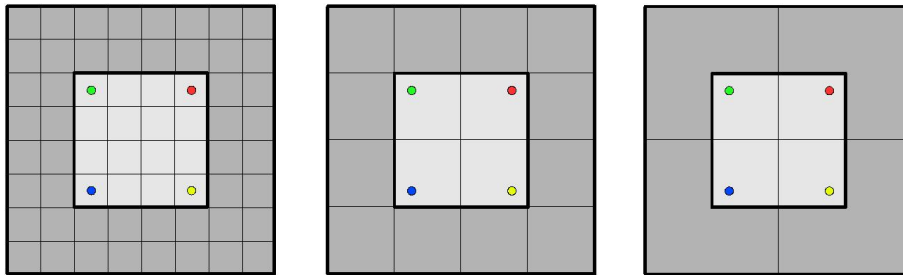
So we have an additional cost of $2b(D_{width} + D_{height}) + 4b^2$. In our case $b \in [0, 2]$, so the maximum size penalty is approximately four times the sum of width and height of a chart. We are sacrificing spatial cost to transparently integrate the method in the graphical pipeline. But as we said, this amount of extra size will not be critical as long as the maximum mip-levels per atlas is two.

This padding enables us to select the available mip-levels depending on the border. Let $M \in \mathbb{N}$ be the number of mip-map levels supported (where 0 is the base texture and M is the lowest mip-level) and $f_b(M)$ a function that returns the border used for M mip-levels, we have in general that:

$$f_b(M) = \begin{cases} 0 & M = 0 \\ 2^{M-1} & M > 0 \end{cases} \quad (3)$$

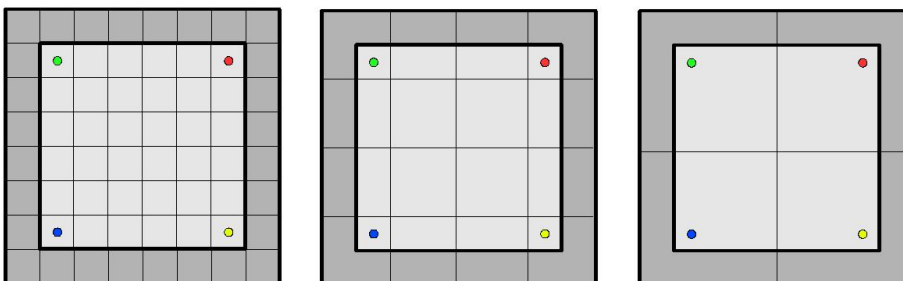
If we do not have mip-maps no border will be required thanks to the clamp to edge technique. If we have one mip-map a border of one texel will be enough. With two mip-levels we will use a border of two texels. We want that the interpolation in the lower mip-levels takes only the colours of the chart (not the neighbours). As we see in the Figure 21, we have a texture with 4×4 pixels and we also want to support two mip-levels, so we add a border with two pixels. As we reduce the resolution through the lower mip-levels the corner points clamped to the edge never reaches further from the dead-center. This property ensures us that the final interpolated colour of the borders will always take the proper colours.

Figure 21: Correct bilinear filtering scheme in a 16x16 chart with 2 border texels and 2 mip-levels



We see in the Figure 22 an incorrect bilinear filtering scheme using only one border and storing two mip-levels. In the first mip-level reduction the corners still get valid colours, but at the lowest mip-level the corners will take colours from neighbouring charts, giving us incorrect results.

Figure 22: Incorrect bilinear filtering scheme in a 16x16 chart with 1 border texel and 2 mip-levels



Finally we have that the required border for a given mip-level M is given by $f_b(M)$ and is the result of a power-of two exponential function (see Equation 3).

8.1.6 Texture atlas support for DXTC compression

The DXTC system introduced in Section 2.1 provides a method to compress textures and decompress them in real-time. Furthermore, graphical hardware has an optimized support to make this decompression faster. Thanks to these advantages, we decided to use it to encode our texture atlas using a DXTC compression scheme.

But a problem appeared when we enabled one or two mip-levels per atlas. Instead we applied the constraints described at Section 8.1.4 smaller artifacts appeared in some boundaries of the textures. However in most of the cases the error was somewhat imperceptible, we tried to find the underlying problem. That problem is clearly focused on the compression stage as we do not have any boundary error without using DXTC compression.

If we analyze the system used for compression by DXTC, we see that it converts a 4×4 block of pixels to a 64-bit or 128-bit quantity resulting in a lossy compression algorithm. So we have a lossy mixing for each 4×4 cell of the image. In the case of DXT1 (used to encode only colour) we store 16 input pixels in 64 bits of output, consisting of two 16-bit RGB 5:6:5 colour values and a 4x4 two bit lookup table. In the decompression stage, if the first colour value c_0 is numerically greater than the second colour value c_1 , then two other colours are calculated, such that:

$$c_2 = \frac{2}{3}c_0 + \frac{1}{3}c_1 \quad c_3 = \frac{1}{3}c_0 + \frac{2}{3}c_1$$

Otherwise, if $c_0 \leq c_1$ then:

$$c_2 = \frac{1}{2}c_0 + \frac{1}{2}c_1$$

And c_3 is transparent black corresponding to premultiplied alpha format. The lookup table is then consulted to determine the colour value for each pixel, with a value of 0 corresponding to c_0 and a value of 3 corresponding to c_3 . The DXT1 does not store alpha data enabling higher compression ratios. So taking account that the cell used for the block decomposition has a dimension 4×4 we introduce the next additional condition: for every mip-level and chart, the block decomposition only affects each chart so the mixing is never done between neighbour charts. This ensures us that the boundaries will not have artifacts because there is no incorrect interaction between charts.

So instead to having a dimension multiple per four it will be different depending on the maximum available mip-level M . What we want is to have in the lowest mip-level a set of charts with a dimension multiple and not less of four. Let $f_d(M, \alpha)$ be a function that returns the possible dimension of a chart given a maximum mip-level M and a scale factor α we have that:

$$f_d(M, \alpha) = 4\alpha(M + 1) \quad \alpha \geq 1 \tag{4}$$

We see that this is a constraint less flexible than the introduced in Section 8.1.4. If we do not have mip-maps ($M = 0$) the dimension constraint will be those introduced by the coordinate compression (multiple of four), but for a maximum mip-level of one or two we have the condition to having a dimension multiple of eight and sixteen respectively. Instead for two levels the restriction is hard than having a power-of two dimension, we have less dimensions available for each chart, then loosing some compression power. However, there are two points that make them more preferable than to do not use texture compression:

1. Higher compression ratios varying between 4:1 to 8:1 that fully compensates the cost to truncate to a valid size of the charts.
2. Thanks to the hierarchical scheme (see Section 8.2) a maximum number of two mip-levels are required for each texture atlas node of the quadtree. This reduces the strenght of the limitation described in Equation 4 to make suitable the DXTC compression for the atlas.

8.2 Real-time visualization

We developed a method to visualize a huge collection of buildings with several textures in real-time using our space-optimized texture atlas. Most of the performance gain comes from the fact that texture batching is heavily decreased. Furthermore, we can guarantee that this batching will never reach a given threshold thanks to the space subdivision. In the next sections, we will present the hierarchical representation of the set of buildings and the level of detail technique used for the texture mapping of repetitive details.

8.2.1 Texture atlas tree

A texture atlas tree, introduced by Buchholz and Döllner [6] defines a quadtree subdivision of the scene in the x-z-plane and a corresponding hierarchy of texture atlases. Each node represents a part of the scene geometry and stores its bounding box. The atlas of an inner node contains downscaled versions of its child nodes. The scene geometry is stored in the leaf nodes. For each frame, the tree provides a small collection of texture atlases containing each visible texture of the scene at an appropriate resolution. The computation of the necessary texture resolution is explained in more detail in Section 8.2.2.

Our texture atlas compression algorithm has been designed to be used in combination with a hierarchical subdivision of the scene. In the case of urban models, we use a quadtree encoding a hierarchy of multiresolution texture atlases. This hierarchy can be seen as a coarse-level collection of mipmapped texture atlases (we say it is coarse-level because all primitives associated with a quadtree node share the same “mipmap” level). The geometry associated with each quadtree node is rendered using a pre-filtered texture atlas whose charts have been scaled down so that their size approximately matches the size of the screen-projection of the polygons they are mapped onto (see Section 7.1), under the viewing conditions causing the quadtree node to be selected for drawing. We use a texture atlas tree to represent all the buildings and consider that they are placed in a two dimensional space and classified by the building height. Let B be a set of buildings and n the maximum deep of the subdivision, we define a criteria that classifies each building of B in a different quadrant taking account the proximity of the quadrant center. The Algorithm 9 referred on the annex clearly illustrates this process.

Figure 23: Hierarchical texture atlas representation



However there are some differences between the texture atlas tree proposed by [6] and our system. Their scheme requires that all the atlas have equal size but our system does not have this limitation. Furthermore, with their solution two sets of texture coordinates are specified for the triangles, one

referring to the original textures, and the other referring to the texture atlas of the leaf node: we only use one set of compressed textures (see Section 8.1.2) per atlas.

In the Figure 23 we see an example of a texture atlas tree using a bin-tree hierarchy. In the upper level N we have a texture atlas containing the subsampled textures of the level $N - 1$. As we see, this system provides a level of detail technique that tries to minimize the texture batching lack. We also see that the total number of texture atlas required for a quadtree with deep n is $\sum_{i=0}^n 4^i$.

For each node we store a value that defines the area precision (meters/texel) of the associated texture atlas. In the preprocessing stage, where we build each texture atlas, we use this area precision to scale the charts to an appropriate dimension. We first define the area precision of the leaf nodes of the texture atlas tree and then recursively set the upper levels with a lower area precision. Let $L \in [1..N_r]$ be the current level in the atlas hierarchy (where $L = 1$ refers to the leaf nodes and $L = N_r$ to the global root node), λ_l the area precision selected for the level 1 and $\lambda(L)$ a function that returns the area precision for a given level L . We set this function as:

$$\lambda(L) = \alpha^{L-1} \lambda_l$$

Where $\alpha > 1$ is a scale factor of the subsampling force. The clever selection of the deep tree N and the value of α is a key to achieve better visual results and performance. We also have to consider the mipmapping when selecting α : different mip-levels may be applied to a level L considering the area precision $\lambda(L + 1)$ of the upper level.

For our implementation, we set $\alpha = 8$, $N_r = 2$ and $\lambda_l = 0.05$. That means that the highest precision is set to 0.05 m/texel and we do not subdivide more than two levels on the quadtree. The α factor progressively subsamples lower resolution levels multiplying by eight the area precision.

Buchholz and Döllner also add a texture border area (see Section 21) to the required texture area to avoid mixing adjacent textures in downsampled versions of the texture atlas. But for their implementation they use a constant border width of eight texels guaranteeing the avoidance of texture pollution for the first three down-sampling levels. Indeed, that supposes heavy waste of memory. So we decided to minimize the amount of memory used for the mipmapping, and designed an adaptive scheme capable to use texture atlas with different mip-levels. We will need more mip-levels in the texture atlas with highest area precision than for example in the texture atlas of the parent node, where in our implementation reaches a 3.2 meters/texel area precision. Let $Mip(L)$ be the number of mip-levels associated to a level L of the tree, we can limit the necessary number of mip-levels seeing how the mipmapping subsampled is done: the reduction is done dividing by two the dimensions of the texture atlas. So we define $Mip(L)$ as:

$$Mip(L) = \begin{cases} 0 & L = N \\ \log_2(\alpha) - 1 & L < N \end{cases}$$

For the node with lowest resolution we do not need any mipmap. For the next nodes with more precision, $Mip(L)$ is defined by the scale factor between the lower and upper level of resolution. We see that this scale is in fact the scale factor $\alpha = \frac{\lambda(L+1)}{\lambda(L)}$. So in our case we have a fixed requirement on the number of mip-levels for all the levels ($Mip(L) = 2$) except for the lowest resolution level. So we are storing the pyramid of subsampled texture atlas explicitly in the quadtree hierarchy and implicitly in the mip-levels associated to each texture atlas. Table 2 shows the resulting $\lambda(L)$ values associated by each L level, and the mip-level deep $Mip(L)$ used.

Table 2: Area precision and mip-level associated to each atlas tree level of our implementation

L	$\lambda(L)$ (m/texel)	$M(L)$
0	0.05	2
1	0.4	2
2	3.2	0

Indeed, we have seen that is enough to assign $Mip(1) = 1$ for the middle level, because the filtering artifacts are less noticeable than in the highest resolution level, where we use two mip-levels.

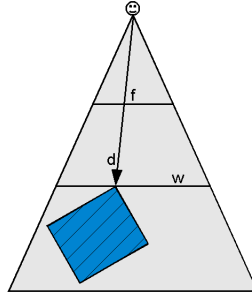
8.2.2 Building rendering

The rendering is performed by a top-down traversal of the texture atlas tree. For each traversed node we perform a view frustum test: if the bounding box is completely outside the view frustum the node is skipped. If it is visible, we compute the point of the bounding box node nearest to the viewer and the distance d to this point. This information is used to calculate the screen projection S_{proj} given an area precision λ . For simplicity, we assume identical scale factors for both screen axes (see Figure 24). The screen projection of a line segment of the length world viewed from the distance d has maximum size if it is oriented orthogonal to the viewing direction. Let w be the viewport width, and f the field of view angle in the y direction, we obtain a projection factor:

$$S_{proj} = \frac{\lambda w}{2d \tan\left(\frac{f}{2}\right)} \quad (5)$$

If S_{proj} is less or equal than a texture resolution threshold t_{res} (texel/pixel), the texture atlas of the current level of resolution is binded and used to wrap the textures. The texture resolution is chosen in a way that the texel-per-pixel ratio is always near to one ($t_{res} = 1$) in order that the amount of necessary texture data remains diminished. However the texel-per-screen pixel ratio can be increased to decrease the texture quality and increase the performance.

Figure 24: Screen projection factor scheme



The quadtree subdivision guarantees that for a given maximum deep, we also have a maximum texture binding switches. In general for a deep n the maximum texture batches is from the order of 4^n . So if we have two maximum levels of deep of our quadtree, the maximum number of texture batches is only sixteen. In some applications with several textures it means a very significant performance gain. Furthermore, for huge environments this batching bound is rarely reached because it only occurs when we are seeing all the scene and we require the maximum resolution for each node.

8.2.3 Terrain visualization

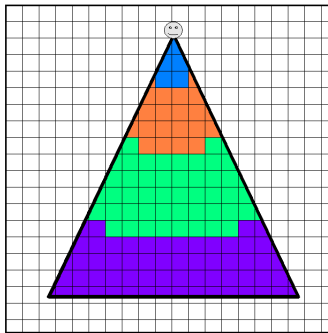
We render the terrain using aerial photos projected on a planar surface. We use a regular partition subdivision of the terrain in tiles. For each tile we have associated a collection of photos with different resolutions that represents the tile region. To increase the performance, we use a quadtree where the leafs are the tiles and the root represents the whole terrain. The rendering is performed by a top-down traversal of the tile tree. As with building rendering (see Section 8.2.2) for each traversed node we perform a view frustum test and discard it if it is completely outside from the view frustum. When we reach the visible leafs, we have to select the appropriate resolution of the texture. We also use the S_{proj} (see equation 5) with $\lambda = 1$ as we want to determine the screen projection of one meter of the

tile at the minimum distance d . Let t_{length} be the length of the tile and v_y the vertical component of the vector joining the viewer and the nearest point of the tile. Then, the required resolution S_{res} in pixels is defined as:

$$S_{res} = S_{proj} t_{length} v_y$$

The value S_{res} give the required resolution for a distance and viewer orientation. To take advantage of the graphical pipeline we recommend to use power-of two textures to represent the different levels of detail of each tile. In our implementation, we used for the highest level of detail a resolution of 512×512 and the successive downsampled versions until the 32×32 resolution. In the Figure 25 we can see a simple 2D scheme of the level of detail used in terrain rendering. The nearest tiles use higher resolution tiles and as we get far lower resolution textures are used.

Figure 25: Terrain rendering LOD example



Highest to lowest resolution: blue, orange, green and purple

One advantage to load separately each resolution version of a tile (instead to store implicitly in a mipmap pyramid) is that at each rendering time we only load the necessary texture tiles. We use a simple texture cache scheme that loads onto the memory the used texture tiles and unloads them when the cache reaches a predefined maximum amount of memory or number of texture units.

Part V

Results

9 Test specifications

9.1 Test model

The model we used to test our technique covers an area of 234.4 km^2 of Barcelona. The Tables 3 and 4 show the geometry and texture information respectively of the test model. The Figure 26 shows a set of thumbnails of the facade textures.

Table 3: Test model geometry information

<i>Triangles</i>	3,266,469
<i>Vertices</i>	6,307,109
<i>Buildings</i>	93,111
<i>Building blocks</i>	13.879
<i>Terrain area</i>	108.43 km^2
<i>Total surface area (facades+ceilings)</i>	234.4 km^2

Table 4: Test model texture information

<i>Number of distinct textures</i>	23,939
<i>Memory space without compression</i>	18 GB (3 bytes/texel)
<i>Memory space with DXT1 compression</i>	3 GB (0.5 bytes/texel)
<i>Memory space with JPEG compression</i>	0.5 GB (36:1)
<i>Average resolution</i>	$0.5 \text{ cm}^2/\text{texel}$
<i>Needed texture memory space for all the city without wrapping</i>	4.2 Tera texels
<i>Needed memory space for all the city without compression</i>	12.6 TB (3 bytes/texel)
<i>Needed memory space for all the city with DXT1 compression</i>	2.1 TB (0.5 bytes/texel)

Figure 26: Thumbnails of a set of Teletlas textures



9.2 Hardware tested

The hardware used for all the tests was:



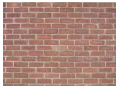





Table 5: Hardware specifications

<i>Graphics card</i>	Nvidia Geforce GTX 260
<i>CPU</i>	Intel Core 2 Quad 2.33 Ghz
<i>RAM memory</i>	3.25 GB

10 Space compression

Space compression results analyzes the image downsampling and compression of a set of eight test images. Also tests the compression achieved by the test model varying the tolerance and the error metric. The specification of the test images are follows:

Table 6: Test image specifications

Name	Dimensions	Snapshot
Window	512x512	(a) 
Rocktile	500x375	(b) 
Bricktile	499x366	(c) 
Fabrictile	400x300	(d) 
Textureatlas	400x400	(e) 
Crayons	516x341	(f) 
Boat	500x336	(g) 
Aerialphoto	521x512	(h) 

10.1 Image downsampling

10.1.1 Downsampling function of test images

The next figures show the associated downsampling function of each test image. The vertical coordinate indicates the metric error obtained and the horizontal one the downsampling direction (width, height and both). The root mean square error is denoted by a red line and the human visual system metric by a green line. The original test image is shown on the lower right region of the figures.

Figure 27: Window downsampling function

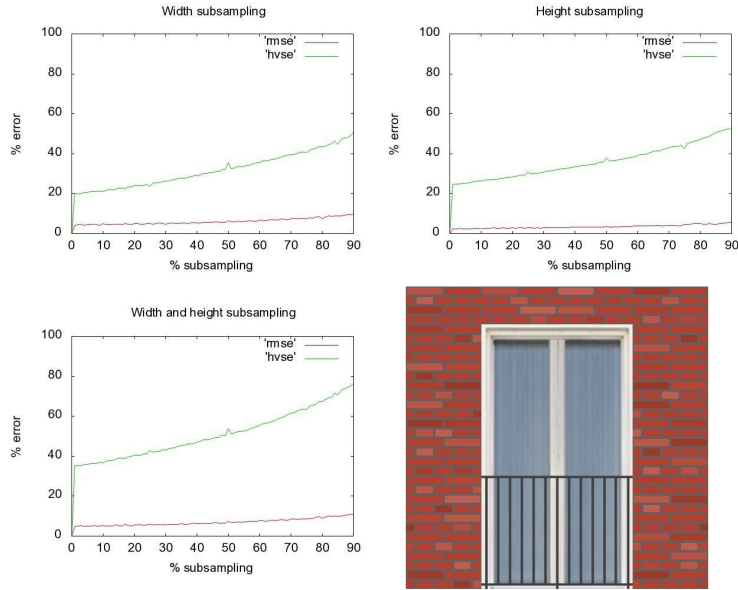


Figure 28: Rocktile downsampling function

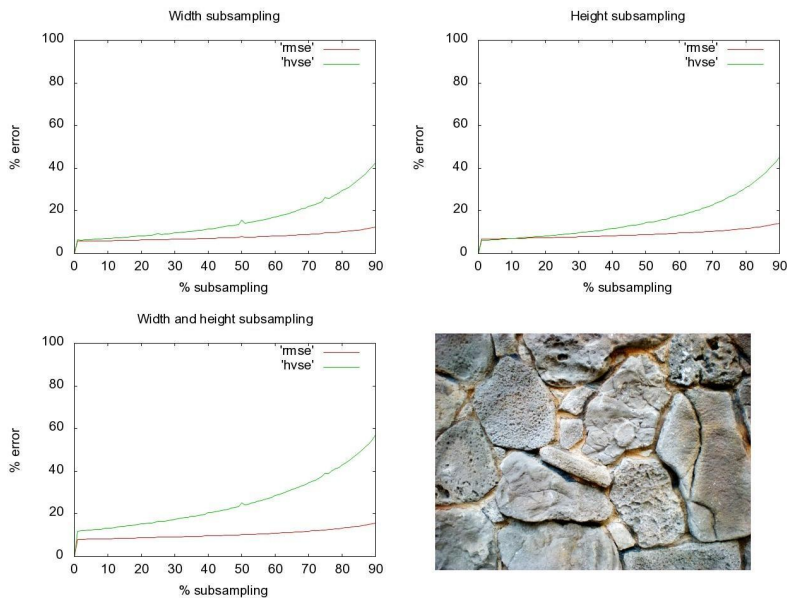


Figure 29: Bricktile downsampling function

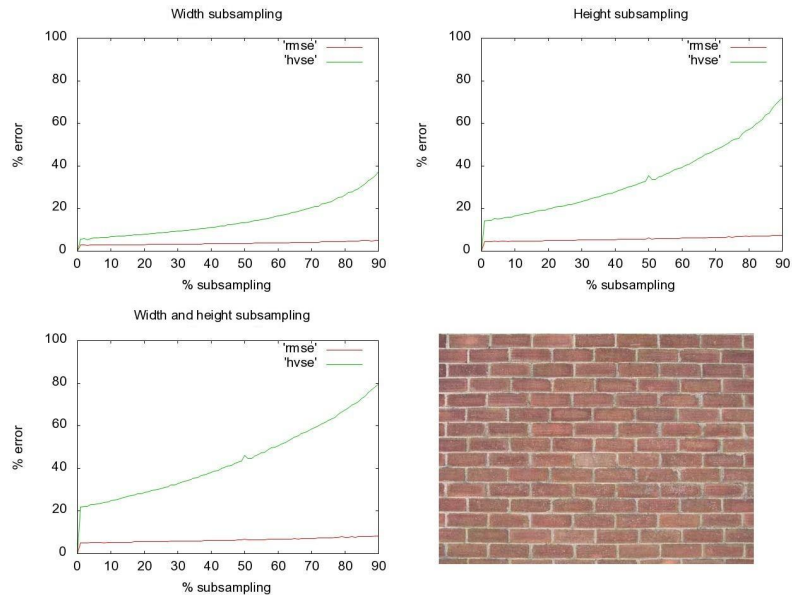


Figure 30: Fabrictile downsampling function

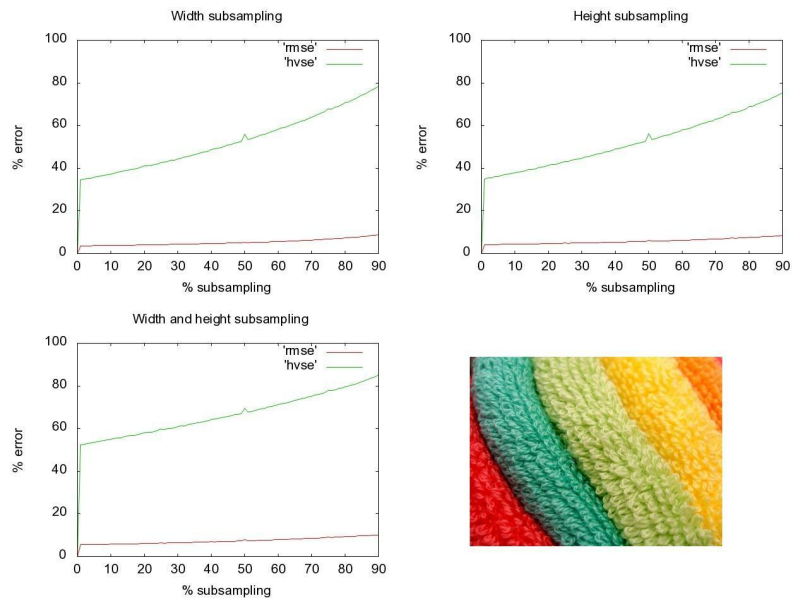


Figure 31: Textureatlas downsampling function

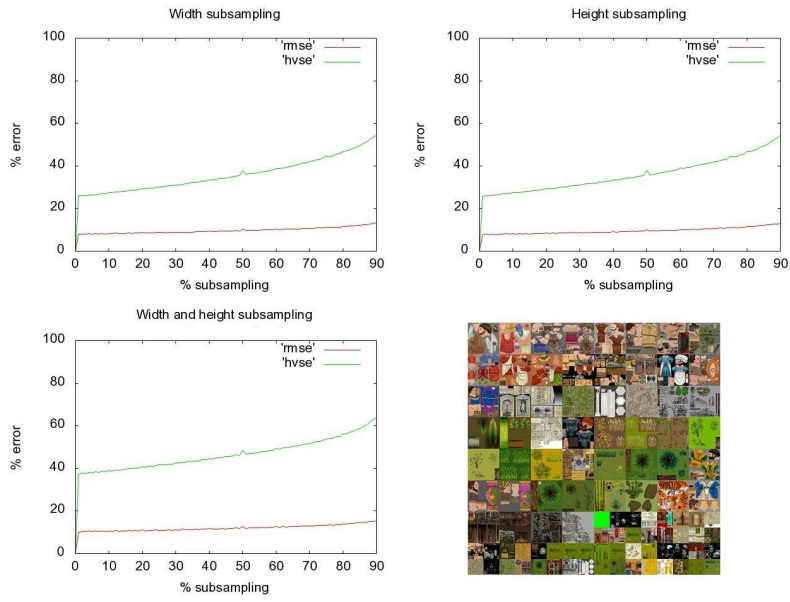


Figure 32: Crayons downsampling function

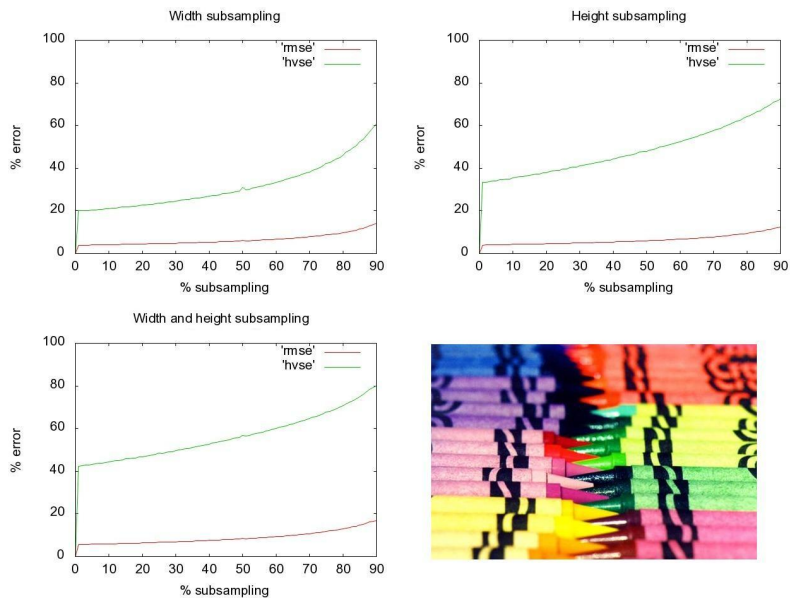


Figure 33: Boat downsampling function

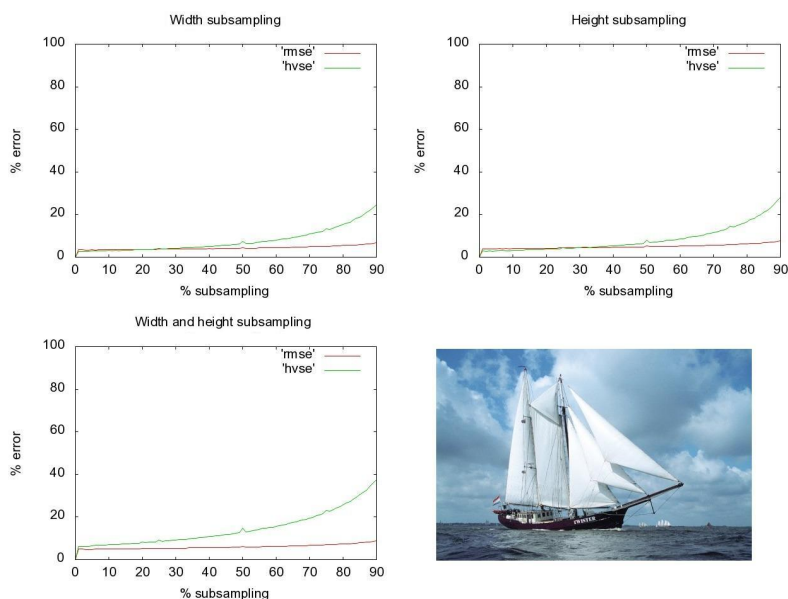
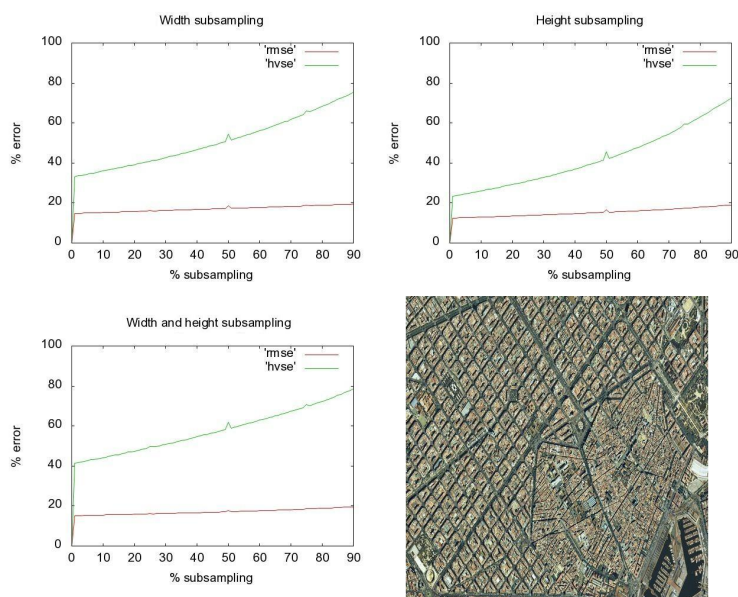


Figure 34: Aerial photo downsampling function



As said in Section 7.2, we see that HVSE has more sensitivity to the effect of downsampling compared with RMSE. That is a desirable property because we want to distinguish between different tolerance levels. RMSE poorly reacts to the downsampling also with a reduction of 90% or greater.

However we also see that HVSE have in some cases high sensitivity applying a little downsampling. That may be a restriction if we define error thresholds not greater than the initial minimum threshold. Per example, in the Figure 30 only reducing by 1% the width and height we obtain more than 50% of error so if we do not have an error threshold greater we will not have any compression. In the Figure

33 we have a different case, all the two metrics start with similar error but another time, HVSE is more sensitive to the downsampling increase.

In general, as depicted in Section 1, we see that RMSE is not suitable in the context of measuring the visual perception of image fidelity: since all image pixels are treated equally image content-dependent variations in image fidelity cannot be accounted for. The Sections 10.1.2 and 10.1.3 also include more tests that demonstrates this poor behaviour.

10.1.2 Reconstruction of test images with varying RMSE visual tolerance

The next figures show in the upper row the test image downsampled and resized to the original size by different factors. The lower rows show, first for a given error metric tolerance L , the downsampled image to match image saliency placed in the bounding box (represents the original image area downsampled to the column factor) and in the next row the image resized to the original size. We can appreciate the increase of the downsampling as we have a bigger error metric tolerance L .

In the Section 10.1.3 we present the same tests using the HVS error metric used by our implementation.

Figure 35: Window reconstruction using RMSE

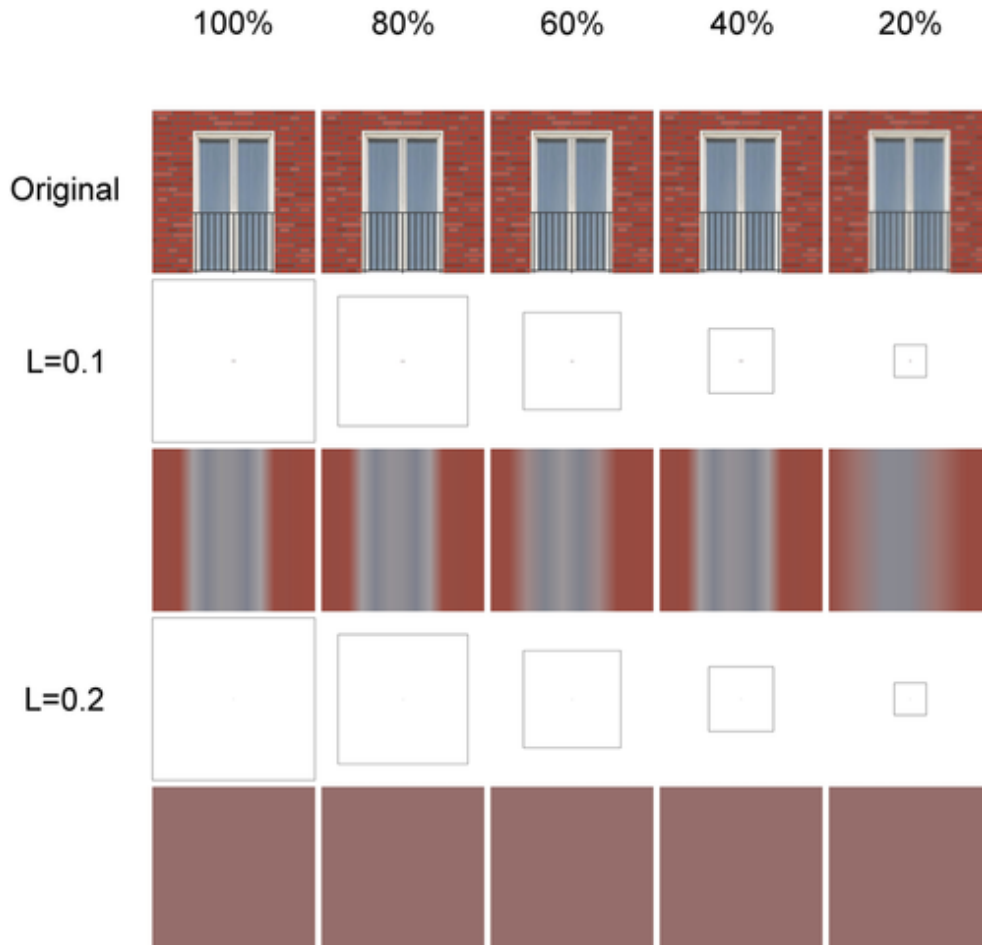


Figure 36: Rocktile reconstruction using RMSE

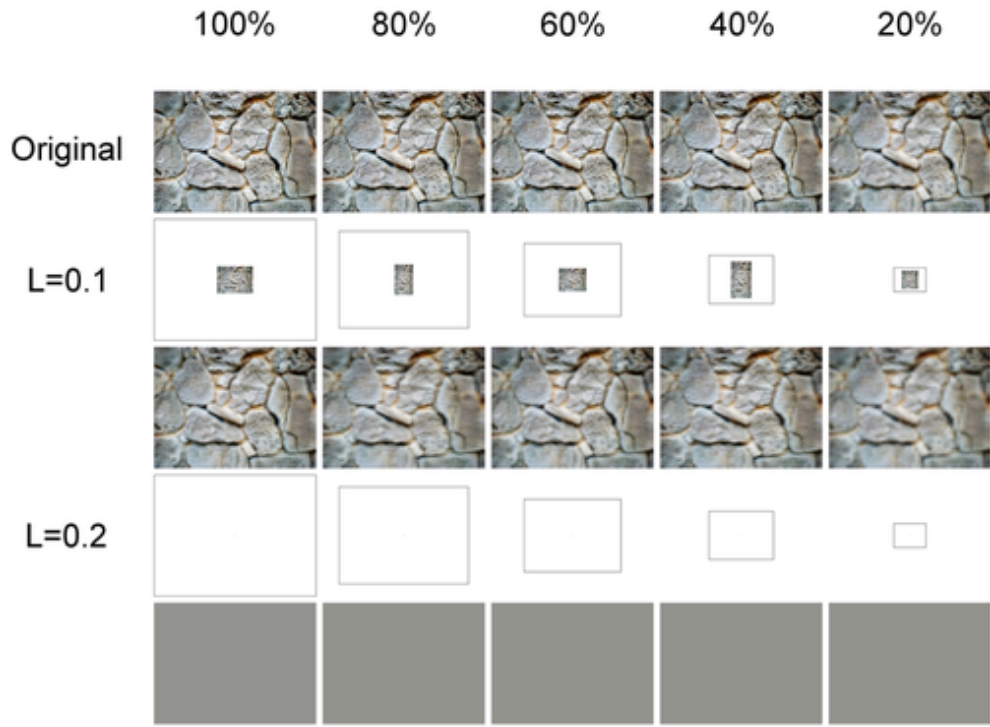


Figure 37: Bricktile reconstruction using RMSE

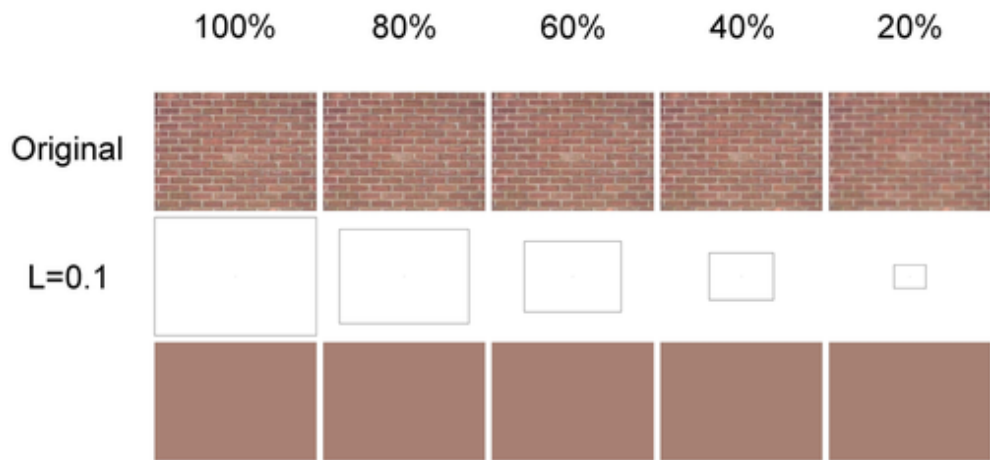


Figure 38: Fabrictile reconstruction using RMSE

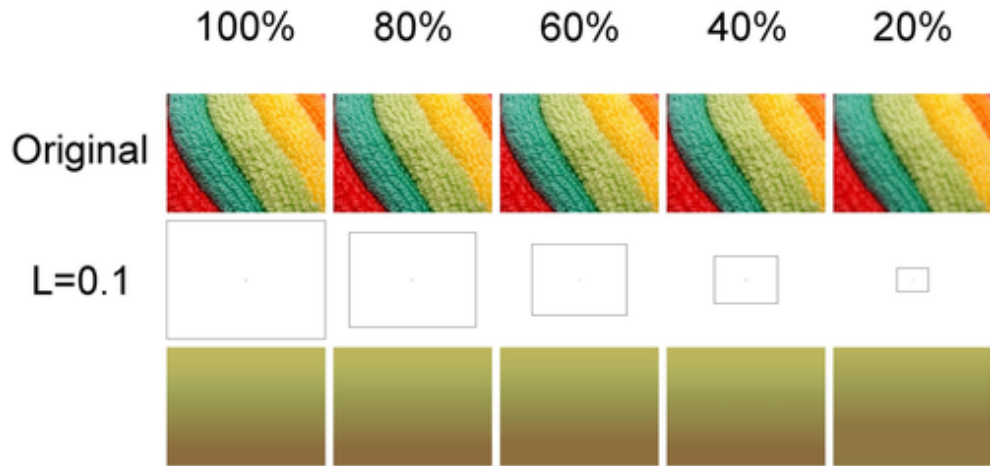


Figure 39: Textureatlas reconstruction using RMSE

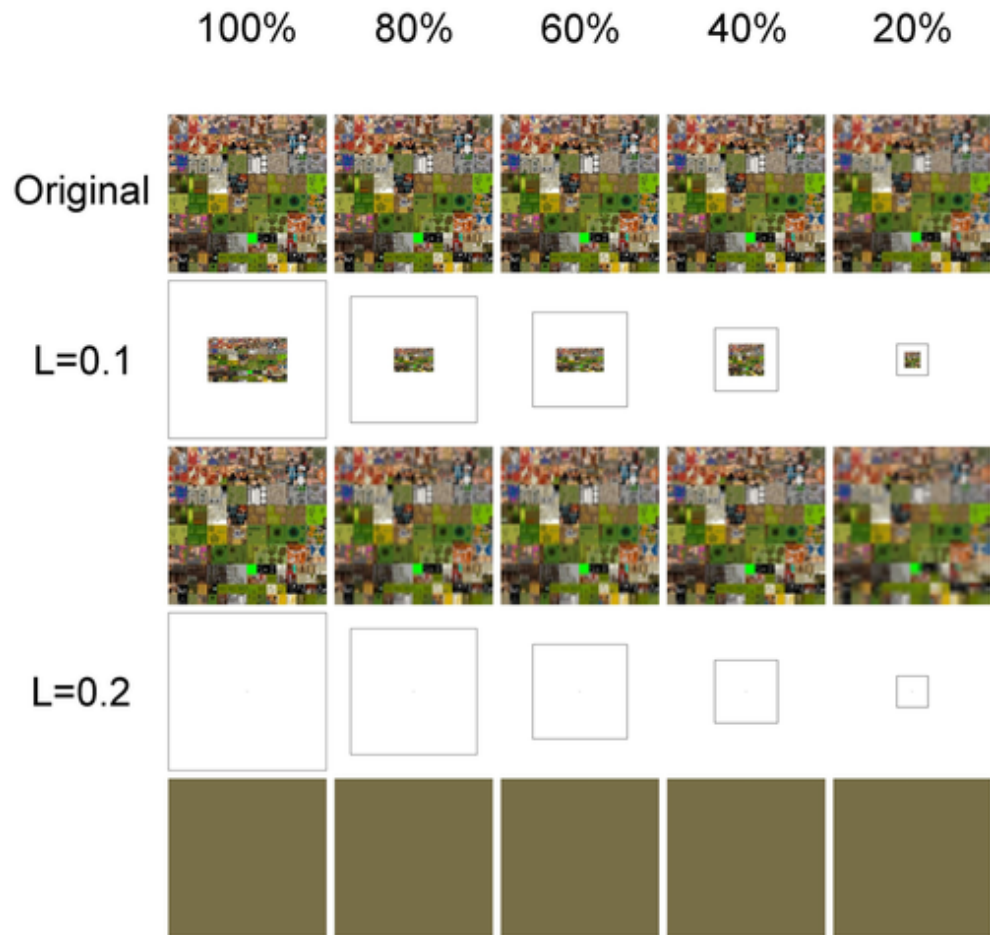


Figure 40: Crayons reconstruction using RMSE

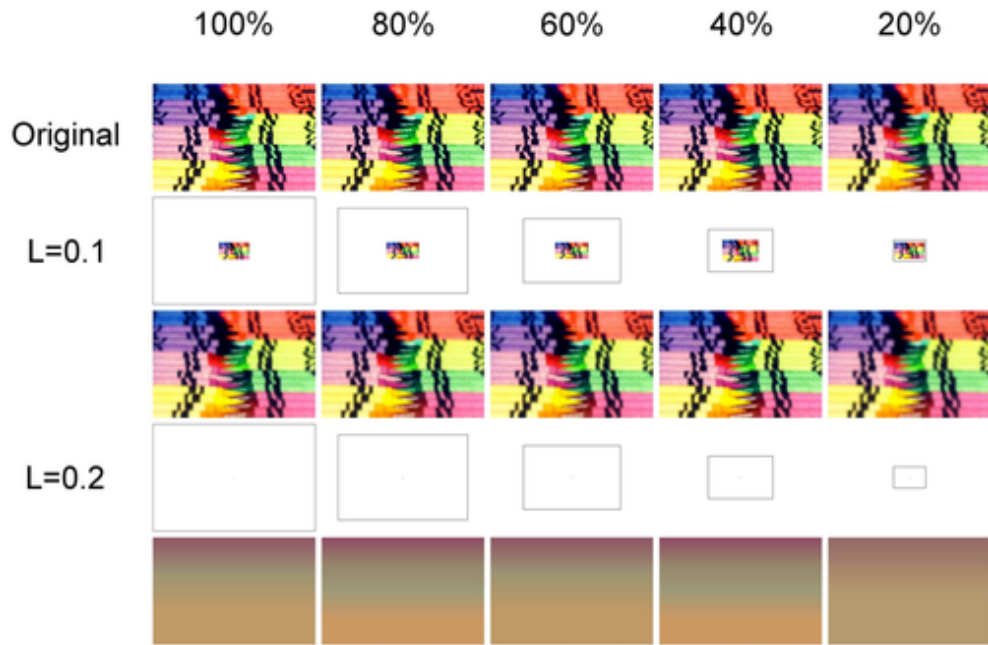


Figure 41: Boat reconstruction using RMSE

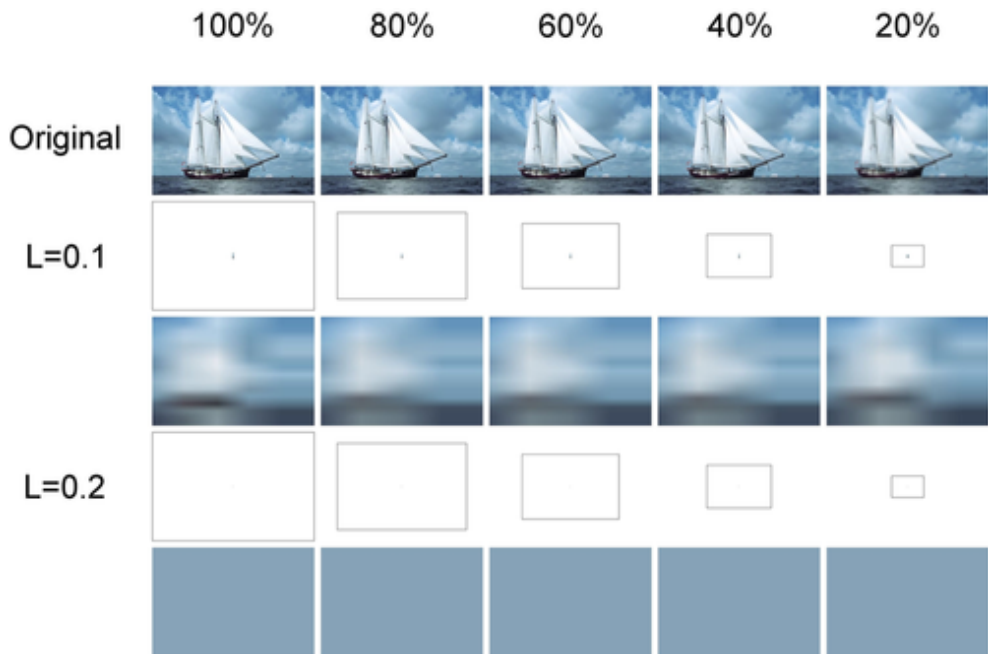
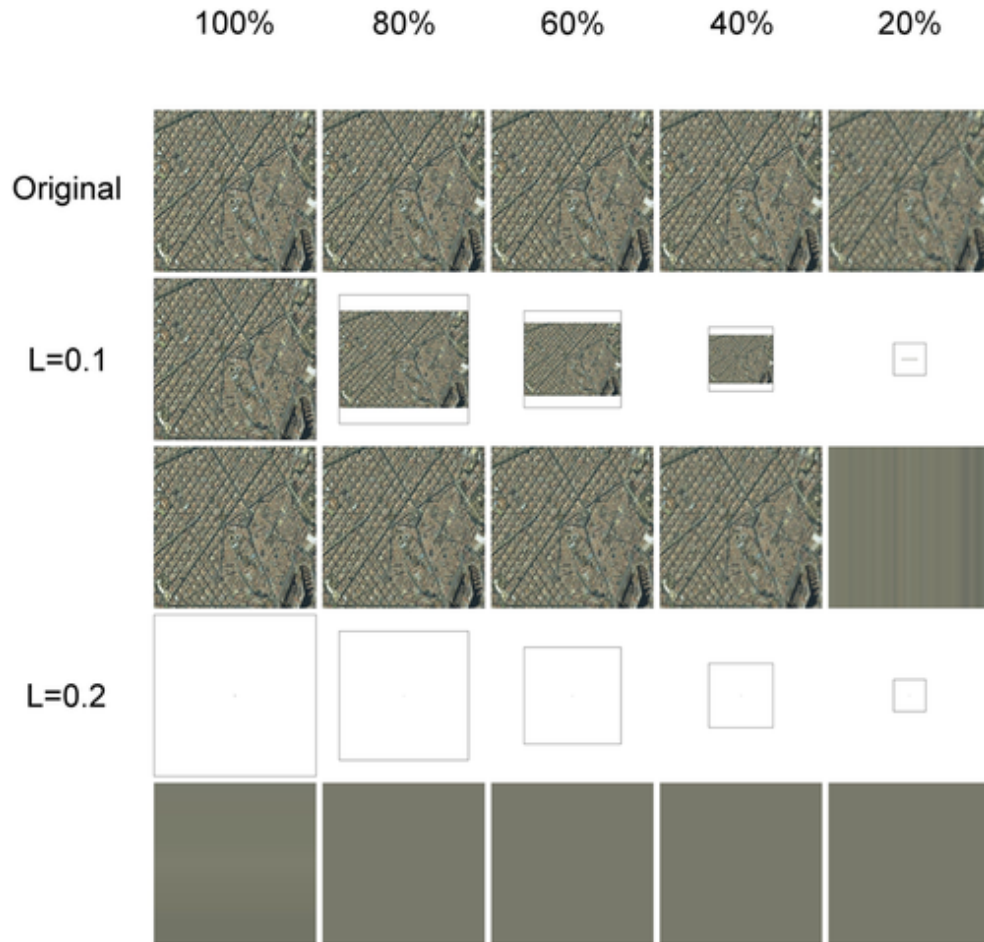


Figure 42: Aerialphoto reconstruction using RMSE



We see that RMSE downsamples too much fast the size of the textures while increasing the visual error tolerance. In some cases, even at 10% of error tolerance we obtain a full downsampling of 1x1 chart size. With a 20% all the tests downsample to 1x1. This implies that we have a limited range of image compression using the method described at Section 7.2.

10.1.3 Reconstruction of test images with varying HVSE visual tolerance

We use the same kind of schemes explained at the Section 10.1.2 to analyze the downsampling varying the visual tolerance. HVS metric is used for our implementation thanks to his better reaction to the downsampling effect and adaptation to the HVS conditions.

Figure 43: Window reconstruction using HVSE

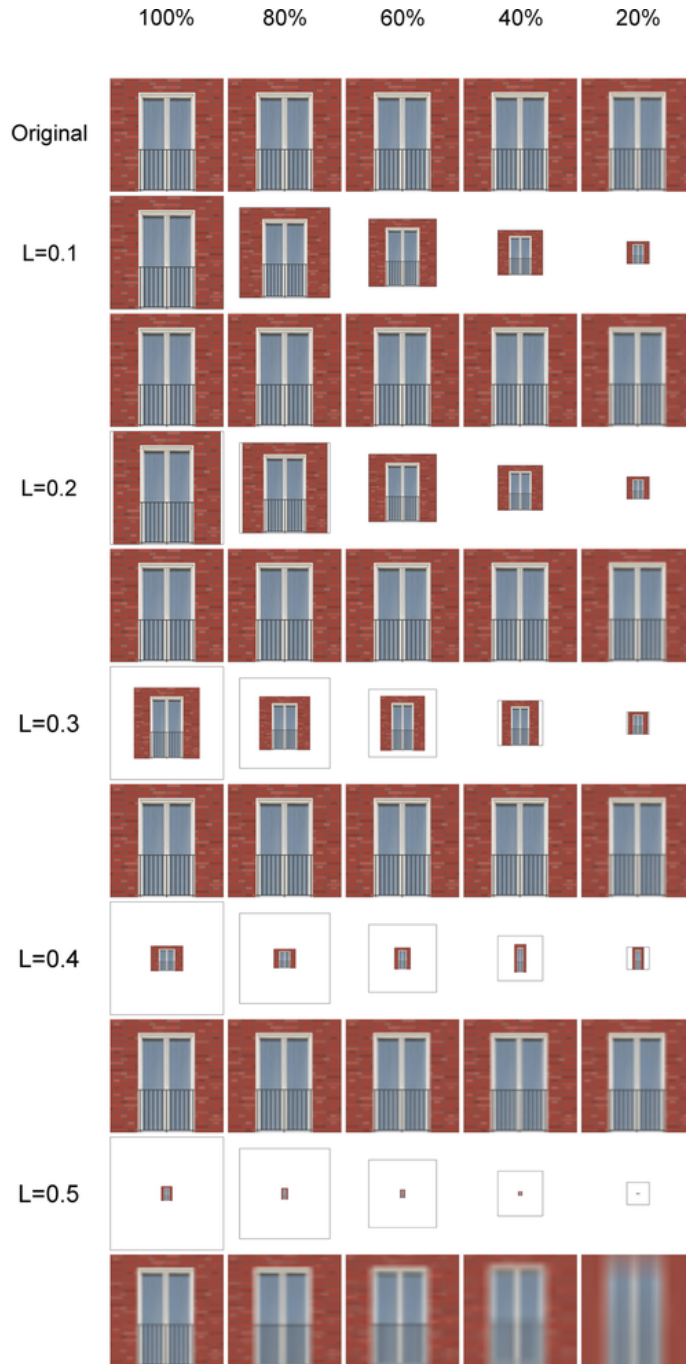


Figure 44: Rocktile reconstruction using HVSE

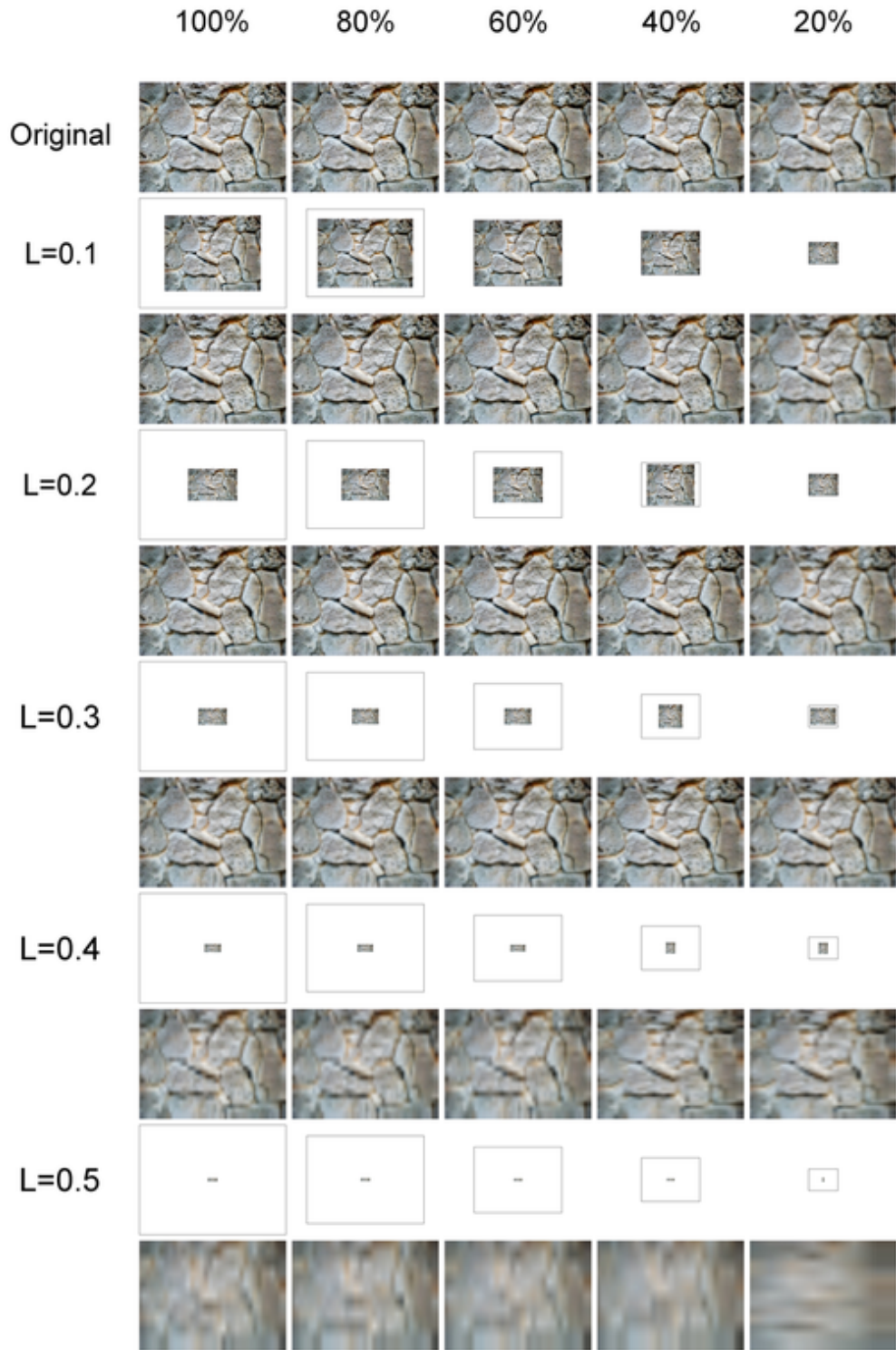


Figure 45: Bricktile reconstruction using HVSE

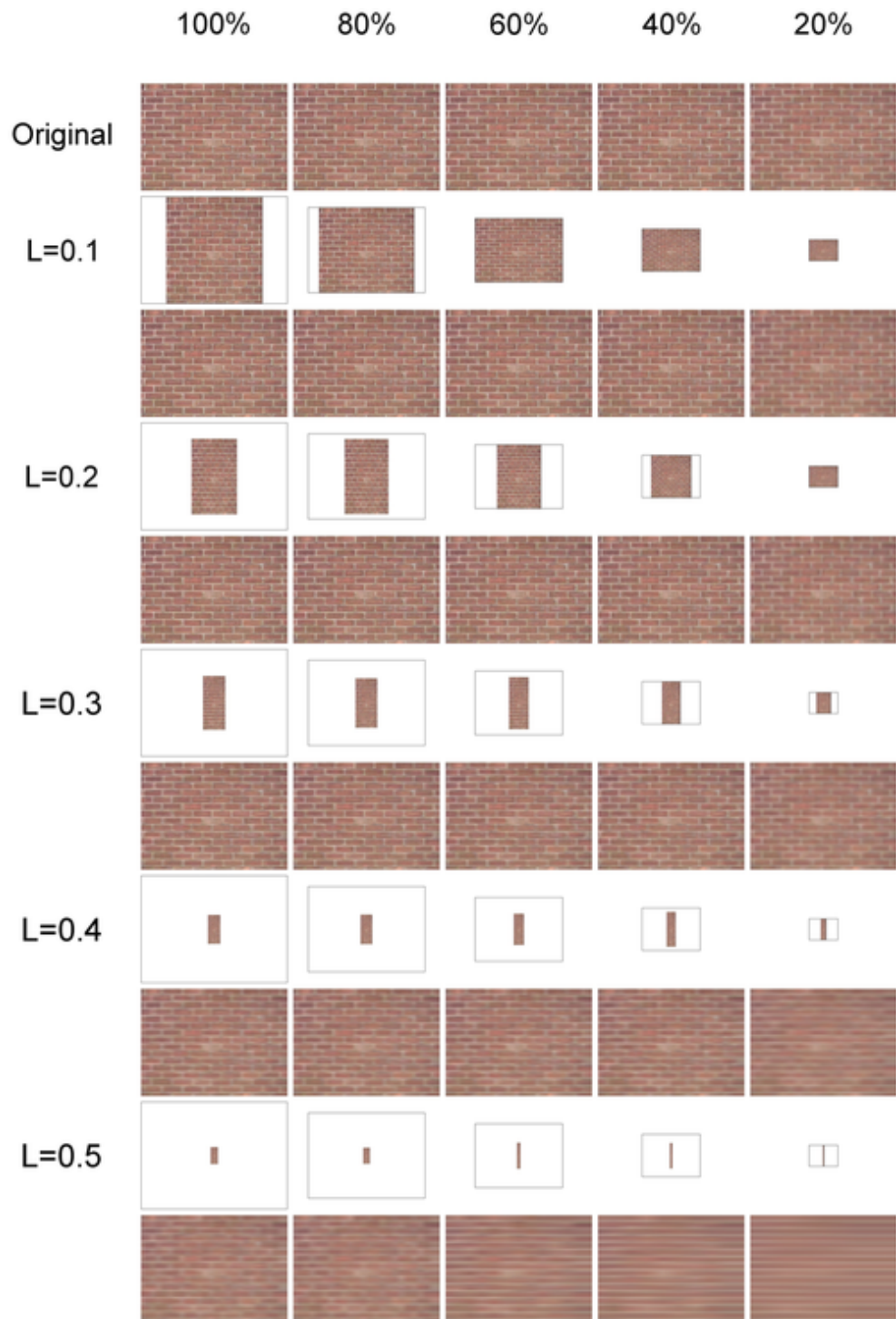


Figure 46: Fabrictile reconstruction using HVSE

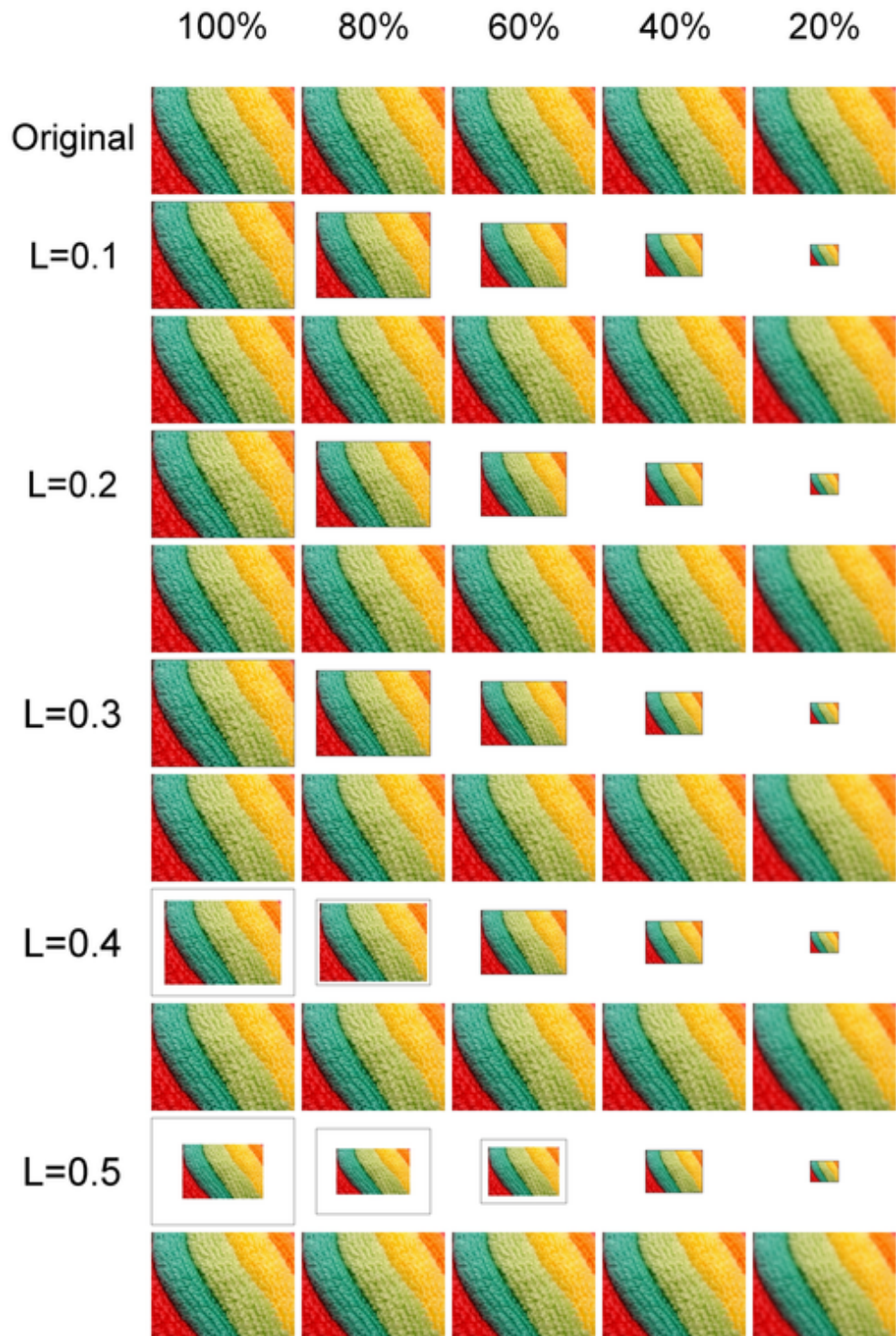


Figure 47: Textureatlas reconstruction using HVSE

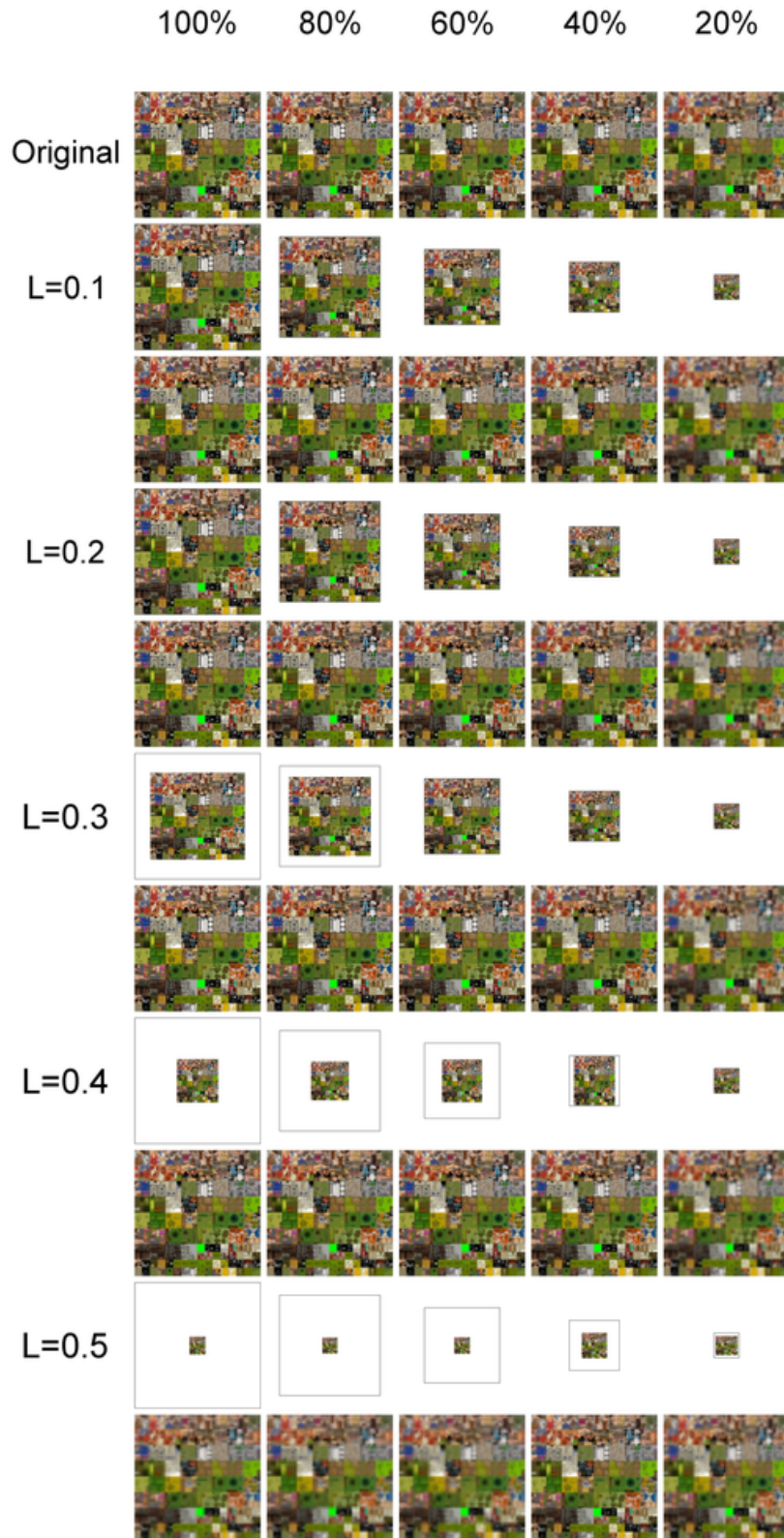


Figure 48: Crayons reconstruction using HVSE

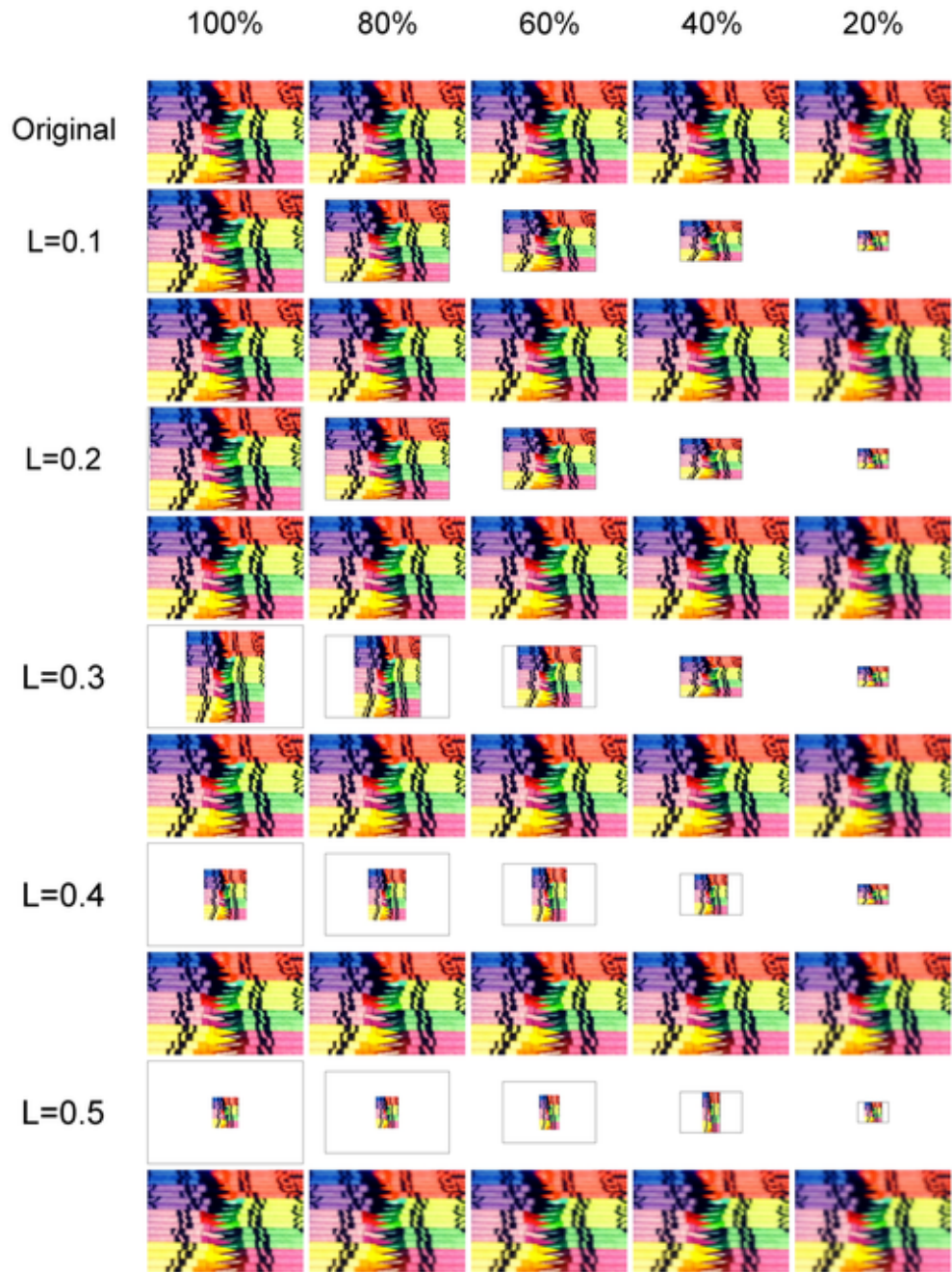
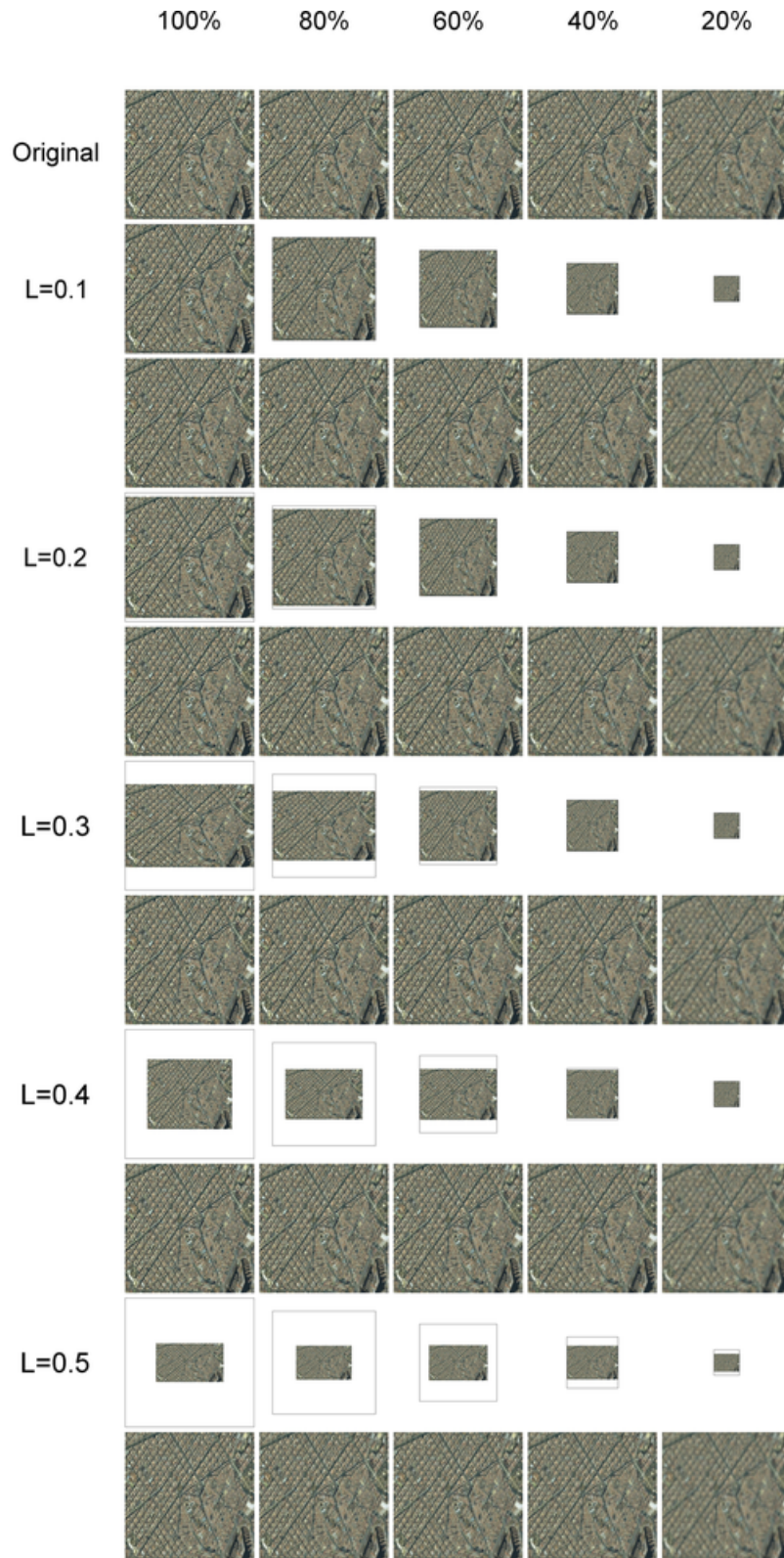


Figure 49: Boat reconstruction using HVSE



Figure 50: Aerialphoto reconstruction using HVSE



We use different L values for the tests of this section and Section 10.1.2 because RMSE slowly reacts to the downsampling and have less error tolerance resolution.

As we see, because HVSE has more sensitivity to the downsampling it provides more resolution in the range of error metric tolerance. That is also clearly illustrated in Section 10.1.1.

Compared with RMSE, that downsamples to a 1x1 chart until it reaches the 20% error tolerance, we have more range using HVSE. In some cases, the compression starts at tolerance error levels greater than 30-40% (e.g. in the Figure 50), so HVS seems to be more restrictive in the initial steps of the user-defined error tolerance. This happens when the images contain more visual perceptual information (see Section 7.2.3 for detailed explanation of the criteria).

We also detect that the stretching is sometimes anisotropic depending on the downsampling direction (e.g. in the Figure 45) meaning that the downsampling process distincts between the amount of information provided separately by width and height.

10.1.4 Test images compression results

The next tables show for each test image the compression ratios achieved by the image downsampling thanks to the matching to view conditions and image saliency. We consider that each image project on a space of 20x15 meters. Each row identifiers denote:

- **View size:** size after the match to view conditions (see Section 7.1).
- **View CR:** compression ratio due the match to view conditions.
- **Visual size:** size after the saliency matching.
- **Visual CR:** compression ratio with respect to view size.
- **Total CR:** total compression ratio achieved.

Window compression results

		0.05 meter/texel							
View size	400x300								
View CR	2.18:1								
		<i>RMSE</i>			<i>HVSE</i>				
		5%	10%	20%	10%	20%	30%	40%	50%
Visual size	350x75	12x1	1x1	398x298	398x298	398x298	225x290	100x103	
Visual CR	4.57:1	10000:1	120000:1	1.01:1	1.01:1	1.01:1	1.83:1	11.65:1	
Total CR	9.98:1	21845:1	262144:1	2.21:1	2.21:1	2.21:1	4.01:1	25.45:1	

		0.4 meter/texel							
View size	50x37								
View CR	141.69:1								
		<i>RMSE</i>			<i>HVSE</i>				
		5%	10%	20%	10%	20%	30%	40%	50%
Visual size	43x18	6x1	1x1	48x35	48x35	48x35	48x18	37x5	
Visual CR	2.39:1	308.33:1	1850:1	1.10:1	1.10:1	1.10:1	2.14:1	10:1	
Total CR	338.68:1	43960:1	262144:1	156:1	156:1	156:1	303:1	1416:1	

	3.2 meter/texel							
View size	6x4							
View CR	10922.7:1							
	<i>RMSE</i>			<i>HVSE</i>				
	<i>5%</i>	<i>10%</i>	<i>20%</i>	<i>10%</i>	<i>20%</i>	<i>30%</i>	<i>40%</i>	<i>50%</i>
Visual size	6x4	6x4	3x4	6x4	4x4	4x3	4x2	4x2
Visual CR	1:1	1:1	2:1	1:1	1.5:1	2:1	3:1	3:1
Total CR	10922:1	10922:1	21845:1	10922:1	16384:1	21845:1	32768:1	32768:1

Rocktile compression results

	0.05 meter/texel							
View size	400x300							
View CR	1.56:1							
	<i>RMSE</i>			<i>HVSE</i>				
	<i>5%</i>	<i>10%</i>	<i>20%</i>	<i>10%</i>	<i>20%</i>	<i>30%</i>	<i>40%</i>	<i>50%</i>
Visual size	375x298	56x93	1x1	362x262	201x126	104x63	62x30	39x12
Visual CR	1.07:1	23:1	120000:1	1.26:1	4.73:1	18.31:1	64.51:1	256.4:1
Total CR	1.67:1	36:1	187500:1	1.97:1	7.40:1	28.61:1	100.8:1	400:1

	0.4 meter/texel							
View size	50x37							
View CR	101.35:1							
	<i>RMSE</i>			<i>HVSE</i>				
	<i>5%</i>	<i>10%</i>	<i>20%</i>	<i>10%</i>	<i>20%</i>	<i>30%</i>	<i>40%</i>	<i>50%</i>
Visual size	48x35	25x18	1x1	48x35	48x35	43x35	37x18	6x13
Visual CR	1.10:1	4.11:1	1850:1	1.1:1	1.1:1	1.22:1	2.77:1	23.71
Total CR	111.6:1	416:1	187500:1	111.6:1	111.6:1	124.58:1	281.5:1	2403.85:1

	3.2 meter/texel							
View size	6x4							
View CR	7812.5:1							
	<i>RMSE</i>			<i>HVSE</i>				
	<i>5%</i>	<i>10%</i>	<i>20%</i>	<i>10%</i>	<i>20%</i>	<i>30%</i>	<i>40%</i>	<i>50%</i>
Visual size	6x4	6x4	3x4	6x4	6x4	3x2	1x2	1x2
Visual CR	1:1	1:1	2	1:1	1:1	4:1	12:1	12:1
Total CR	7812.5:1	7812.5:1	15625:1	7812.5:1	7812.5:1	31250:1	93750:1	93750:1

Bricktile compression results

	0.05 meter/texel							
View size	400x300							
View CR	1.52:1							
	<i>RMSE</i>			<i>HVSE</i>				
	<i>5%</i>	<i>10%</i>	<i>20%</i>	<i>10%</i>	<i>20%</i>	<i>30%</i>	<i>40%</i>	<i>50%</i>
Visual size	25x150	1x1	1x1	312x298	150x253	73x164	40x103	21x58
Visual CR	32:1	120000:1	120000:1	1.29:1	3.16:1	10.02:1	29.12:1	98.52:1
Total CR	48.1:1	182634:1	182634:1	1.96:1	4.81:1	15.25:1	44.32:1	149.94:1

0.4 meter/texel								
View size	50x37							
View CR	98.72:1							
	<i>RMSE</i>			<i>HVSE</i>				
	5%	10%	20%	10%	20%	30%	40%	50%
Visual size	1x18	1x1	1x1	48x35	48x35	37x35	3x35	1x1
Visual CR	102.7:1	1850:1	1850:1	1.1:1	1.1:1	1.42:1	17.61:1	1850:1
Total CR	10146:1	182634:1	182634:1	108.71:1	108.71:1	141:1	1739:1	182634:1

3.2 meter/texel								
View size	6x4							
View CR	7609.75:1							
	<i>RMSE</i>			<i>HVSE</i>				
	5%	10%	20%	10%	20%	30%	40%	50%
Visual size	6x4	6x4	1x4	6x4	1x4	1x1	1x1	1x1
Visual CR	1:1	1:1	6:1	1:1	6:1	24:1	24:1	24:1
Total CR	7609.75:1	7609.75:1	45658:1	7609:1	45658:1	182634:1	182634:1	182634:1

Fabric tile compression results

0.05 meter/texel								
View size	400x300							
View CR	1:1							
	<i>RMSE</i>			<i>HVSE</i>				
	5%	10%	20%	10%	20%	30%	40%	50%
Visual size	225x187	1x9	1x1	398x298	398x298	398x298	343x248	243x154
Visual CR	2.85:1	13333:1	120000:1	1.01:1	1.01:1	1.01:1	1:1.41	3.2:1
Total CR	2.85:1	13333:1	120000:1	1.01:1	1.01:1	1.01:1	1:1.41	3.2:1

0.4 meter/texel								
View size	50x37							
View CR	64.86:1							
	<i>RMSE</i>			<i>HVSE</i>				
	5%	10%	20%	10%	20%	30%	40%	50%
Visual size	25x18	1x4	1x1	48x35	48x35	48x35	48x35	48x35
Visual CR	4.11:1	462.5:1	1850:1	1.1:1	1.1:1	1.1:1	1.1:1	1.1:1
Total CR	4.11:1	30000:1	120000:1	71.4:1	71.4:1	71.4:1	71.4:1	71.4:1

3.2 meter/texel								
View size	6x4							
View CR	5000:1							
	<i>RMSE</i>			<i>HVSE</i>				
	5%	10%	20%	10%	20%	30%	40%	50%
Visual size	6x4	6x4	3x4	6x3	4x4	4x3	4x3	4x3
Visual CR	1:1	1:1	2:1	1:1	1.5:1	2:1	2:1	2:1
Total CR	5000:1	5000:1	10000:1	5000:1	7500:1	10000:1	10000:1	10000:1

Textureatlas compression results

0.05 meter/texel								
View size	400x300							
View CR	1.33:1							
	<i>RMSE</i>			<i>HVSE</i>				
	<i>5%</i>	<i>10%</i>	<i>20%</i>	<i>10%</i>	<i>20%</i>	<i>30%</i>	<i>40%</i>	<i>50%</i>
Visual size	398x298	125x56	1x1	400x300	400x300	398x298	262x243	131x152
Visual CR	1.01:1	17.14:1	120000:1	1:1	1:1	1.01:1	1.88:1	6.02:1
Total CR	1.34:1	22.85:1	160000:1	1.33:1	1.33:1	1.34:1	2.51:1	8.03:1

0.4 meter/texel								
View size	50x37							
View CR	86.48:1							
	<i>RMSE</i>			<i>HVSE</i>				
	<i>5%</i>	<i>10%</i>	<i>20%</i>	<i>10%</i>	<i>20%</i>	<i>30%</i>	<i>40%</i>	<i>50%</i>
Visual size	48x34	25x18	1x1	50x37	50x37	50x37	48x35	48x35
Visual CR	1.13:1	4.11:1	1850:1	1:1	1:1	1:1	1.10:1	1.10:1
Total CR	98:1	355:1	160000:1	64.86:1	64.86:1	64.86:1	95.23:1	95.23:1

3.2 meter/texel								
View size	6x4							
View CR	6666:1							
	<i>RMSE</i>			<i>HVSE</i>				
	<i>5%</i>	<i>10%</i>	<i>20%</i>	<i>10%</i>	<i>20%</i>	<i>30%</i>	<i>40%</i>	<i>50%</i>
Visual size	6x4	6x4	3x4	6x4	4x4	4x3	3x3	3x3
Visual CR	1:1	1:1	2:1	1:1	1.5:1	2:1	2.66:1	2.66:1
Total CR	6666:1	6666:1	13333:1	6666:1	10000:1	133333.3	17777:1	17777:1

Crayons compression results

0.05 meter/texel								
View size	400x300							
View CR	1.46:1							
	<i>RMSE</i>			<i>HVSE</i>				
	<i>5%</i>	<i>10%</i>	<i>20%</i>	<i>10%</i>	<i>20%</i>	<i>30%</i>	<i>40%</i>	<i>50%</i>
Visual size	350x225	87x56	1x3	400x300	399x298	275x298	153x199	90x121
Visual CR	1.52:1	24.63:1	20000:1	1:1	1.01	1.46:1	3.94:1	11.01:1
Total CR	2.23:1	36.11:1	58652:1	1.46:1	1.48:1	2.14:1	5.77:1	16.15:1

0.4 meter/texel								
View size	50x37							
View CR	95.11:1							
	<i>RMSE</i>			<i>HVSE</i>				
	<i>5%</i>	<i>10%</i>	<i>20%</i>	<i>10%</i>	<i>20%</i>	<i>30%</i>	<i>40%</i>	<i>50%</i>
Visual size	48x36	48x18	1x2	50x37	48x35	48x34	48x34	48x34
Visual CR	1.1:1	2.14:1	925:1	1:1	1.1:1	1.1:1	1.1:1	1.1:1
Total CR	104.73:1	203.65:1	87978:1	95.11:1	104.73:1	107:1	107:1	107:1

	3.2 meter/texel							
View size	6x4							
View CR	7331:1							
	<i>RMSE</i>			<i>HVSE</i>				
	<i>5%</i>	<i>10%</i>	<i>20%</i>	<i>10%</i>	<i>20%</i>	<i>30%</i>	<i>40%</i>	<i>50%</i>
Visual size	6x4	6x4	3x4	6x4	4x4	4x3	4x3	4x3
Visual CR	1:1	1:1	2:1	1:1	1:5	2:1	2:1	2:1
Total CR	7331:1	7331:1	14663:1	7331:1	10997:1	14663:1	14663:1	14663:1

Boat compression results

	0.05 meter/texel							
View size	400x300							
View CR	1.4:1							
	<i>RMSE</i>			<i>HVSE</i>				
	<i>5%</i>	<i>10%</i>	<i>20%</i>	<i>10%</i>	<i>20%</i>	<i>30%</i>	<i>40%</i>	<i>50%</i>
Visual size	100x75	4x11	1x1	143x65	60x22	32x12	7x18	4x12
Visual CR	16:1	2727.27:1	120000:1	12.91:1	90.9:1	312.5:1	952.3:1	2500:1
Total CR	22.4:1	318.18:1	168000:1	18.07:1	127.27:1	437.5:1	1333:1	3500:1

	0.4 meter/texel							
View size	50x37							
View CR	90.81:1							
	<i>RMSE</i>			<i>HVSE</i>				
	<i>5%</i>	<i>10%</i>	<i>20%</i>	<i>10%</i>	<i>20%</i>	<i>30%</i>	<i>40%</i>	<i>50%</i>
Visual size	25x18	12x9	1x1	50x37	25x27	14x13	9x6	4x4
Visual CR	4.11:1	17.12:1	1850:1	1:1	2.74:1	10.16:1	34.25:1	115.62:1
Total CR	373.33:1	1555.56:1	168000:1	90.81:1	248.88:1	923:1	3111:1	10500:1

	3.2 meter/texel							
View size	6x4							
View CR	7000:1							
	<i>RMSE</i>			<i>HVSE</i>				
	<i>5%</i>	<i>10%</i>	<i>20%</i>	<i>10%</i>	<i>20%</i>	<i>30%</i>	<i>40%</i>	<i>50%</i>
Visual size	6x4	6x4	3x4	6x4	4x4	4x3	4x3	3x3
Visual CR	1:1	1:1	7000:1	1:1	1.5	2:1	2:1	2.66:1
Total CR	7000:1	7000:1	14000:1	7000:1	10500:1	14000:1	14000:1	18666:1

Aerialphoto compression results

	0.05 meter/texel							
View size	400x300							
View CR	2.18:1							
	<i>RMSE</i>			<i>HVSE</i>				
	<i>5%</i>	<i>10%</i>	<i>20%</i>	<i>10%</i>	<i>20%</i>	<i>30%</i>	<i>40%</i>	<i>50%</i>
Visual size	400x300	398x262	1x1	400x300	400x300	400x300	312x267	225x196
Visual CR	1:1	1.15:1	120000:1	1:1	1:1	1:1	1.44:1	2.72:1
Total CR	2.18:1	2.51:1	262144:1	2.18:1	2.18:1	2.18:1	3.14:1	5.94:1

0.4 meter/texel								
View size	50x37							
View CR	141.69:1							
	<i>RMSE</i>			<i>HVSE</i>				
	5%	10%	20%	10%	20%	30%	40%	50%
Visual size	25x18	1x1	1x1	50x37	50x37	48x27	43x4	1x4
Visual CR	4.11:1	1850:1	1850:1	1:1	1:1	1.42:1	10.75:1	462.5:1
Total CR	582.54:1	262144:1	262144:1	141.69:1	141.69:1	202.27:1	1524:1	65536:1

3.2 meter/texel								
View size	6x4							
View CR	10922:1							
	<i>RMSE</i>			<i>HVSE</i>				
	5%	10%	20%	10%	20%	30%	40%	50%
Visual size	6x4	6x4	1x4	6x4	3x4	1x1	1x1	1x1
Visual CR	1:1	1:1	6:1	1:1	2:1	24:1	24:1	24:1
Total CR	10922:1	10922:1	65536:1	10922:1	21845:1	262144:1	262144:1	262144:1

We see that RMSE achieves high compression ratios because of its low sensitivity to the downsampling effect, also demonstrated on Section 10.1.1, 10.1.2 and 10.1.3. This commonly supposes a heavy loss of image information for error metric tolerances greater than 10% or even less.

In the case of HVSE, we see in some cases low reaction to downsampling with the lowest tolerance level, but the correlation between the reduction of size and the user-defined error tolerance is higher than RMSE correlation. We find more utility to use compression using HVSE for the levels with the highest area precision. In the test examples, using 0.05m/texel area precision, the compression ratio with respect to the compression to match to view conditions goes from the range of 1:1 to 2500:1, but in the average case reacts much more to the downsampling effect than RMSE.

It is interesting to see that in some cases, the visual size is greater when downsampling the levels with less area precision (e.g. the Aerialphoto using RMSE at 20% and 3.2 meter/texel have a visual size of 1x4, and using 0.05meter/texel just 1x1). This behaviour appears only when using RMSE and is another sign of its poor ability to measure the visual perception of image fidelity.

10.1.5 Image downsampling results of test model

The next table shows the compression results with the test model without using any error metric. The area precision of the levels are the default specified in Table 2.

Table 7: Texture compression with the whole city

	<i>Uncompressed size (MB)</i>	<i>View matching size (MB)</i>	<i>CR View matching</i>
Level 0	622130.85	437460.93	14.22:1
Level 1	622130.85	3860.91	1611:1
Level 2	622130.85	16.41	379000:1
Total	1866395.80	441396.86	42.28:1

We see that the view matching downsampling involves a significant compression of the space used to represent each LOD level. The next six tables show the compression obtained by introducing the image saliency matching. The second row indicates the compression ratio with respect to the view matching size and the third one the total compression.

Table 8: City downsampling RMSE 5%

	<i>Saliency matching size (MB)</i>	<i>CR wrt view matching (MB)</i>	<i>Total CR</i>
Level 0	1732016.76	2.53:1	35:92
Level 1	3811.55	1.01:1	1632:1
Level 2	15.87	1.03:1	391738:1
Total	177016.07	2.49:1	105:1

Table 9: City downsampling RMSE 10%

	<i>Saliency matching size (MB)</i>	<i>CR wrt view matching (MB)</i>	<i>Total CR</i>
Level 0	13079.76	33:1	475:1
Level 1	3337.20	1.16:1	1864:1
Level 2	15.87	1.03:1	391738:1
Total	16432.	26:1	1135:1

Table 10: City downsampling RMSE 20%

	<i>Saliency matching size (MB)</i>	<i>CR wrt view matching (MB)</i>	<i>Total CR</i>
Level 0	90.02	4859:1	69102:1
Level 1	220.42	17.52:1	28224:1
Level 2	15.87	1.03:1	391738:1
Total	326.33	1352:1	57192:1

We see that RMSE obtains too much high compression ratios thanks to his poor reaction to the downsampling effect. In the level with 20% of error tolerance the total CR reaches 57192:1 causing an unacceptable loss of information.

Table 11: City downsampling HVSE 10%

	<i>Saliency matching size (MB)</i>	<i>CR wrt view matching (MB)</i>	<i>Total CR</i>
Level 0	243120.43	1.8:1	25:1
Level 1	3642.16	1.06:1	1708:1
Level 2	15.56	1.05:1	399599:1
Total	246778.16	1.79:1	75:1

Table 12: City downsampling HVSE 30%

	<i>Saliency matching size (MB)</i>	<i>CR wrt view matching (MB)</i>	<i>Total CR</i>
Level 0	56194.18	7.79:1	110:1
Level 1	1449.67	2.66:1	4291:1
Level 2	15.56	1.05:1	399599:1
Total	57659.43	7.66:1	323:1

Table 13: City downsampling HVSE 50%

	<i>Saliency matching size (MB)</i>	<i>CR wrt view matching (MB)</i>	<i>Total CR</i>
Level 0	17174.39	25.48:1	362:1
Level 1	805.83	4.79:1	7720:1
Level 2	15.56	1.05:1	399499:1
Total	17995.80	24.53:1	1037:1

As we see, image saliency matching is also capable to compress the space using different metric tolerances and maintaining the visual fidelity. The total compression ratios go from 75:1 with 10% to 1037:1 with 50% and are more correlated compared with RMSE.

10.2 Packing

The next figures show for different packing sets the texture atlas obtained using four different configurations. An atlas is optimized if it uses the Algorithm 6 and is stretched if uses the Algorithm 5, both presented in Section 7.3.3. A close-up view is also shown in order to appreciate the unused space between the packed charts.

Figure 51: Texture set 1 packing

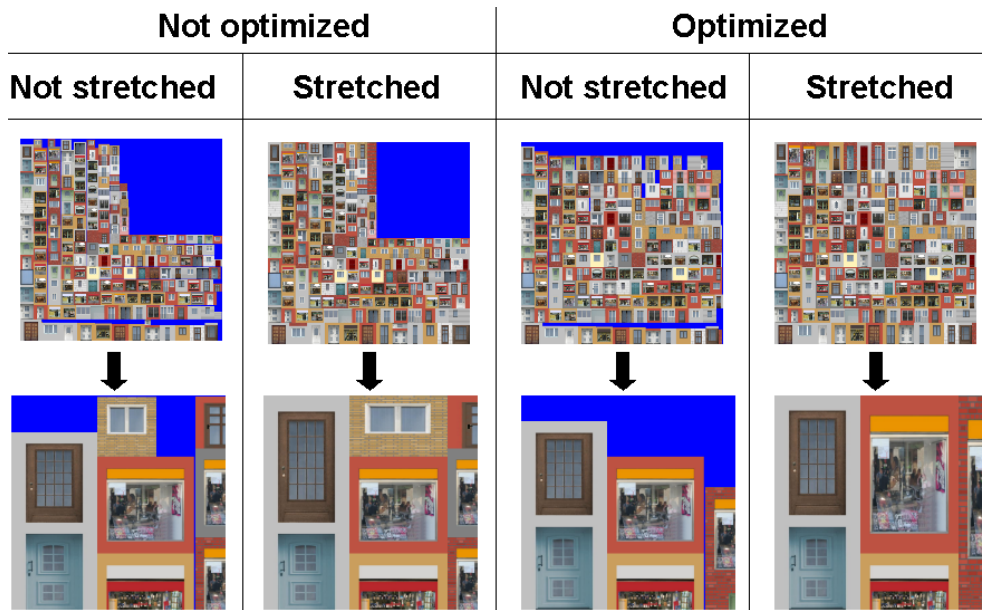


Figure 52: Texture set 2 packing

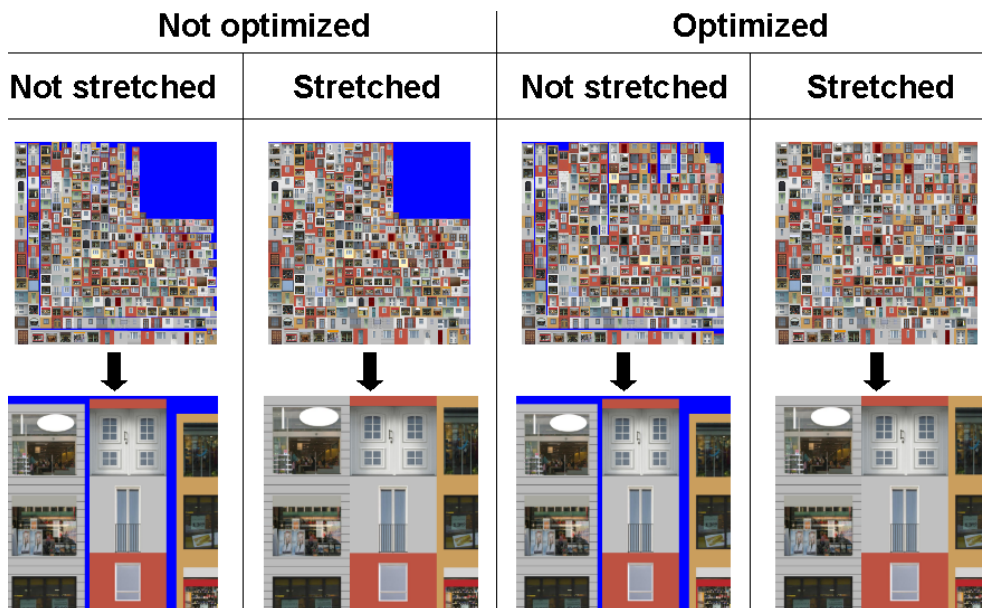


Figure 53: Texture set 3 packing

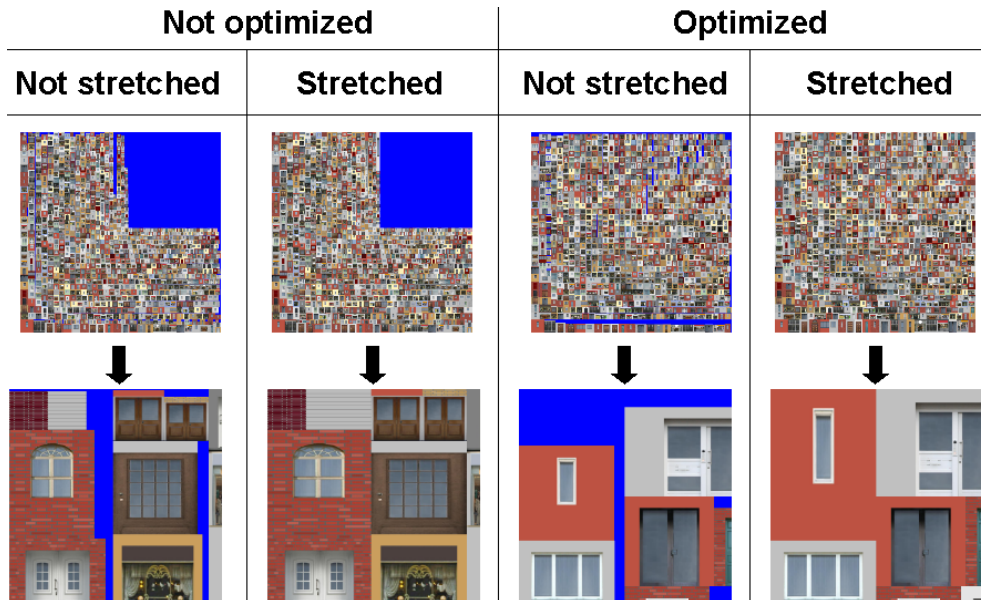


Figure 54: Texture set 4 packing

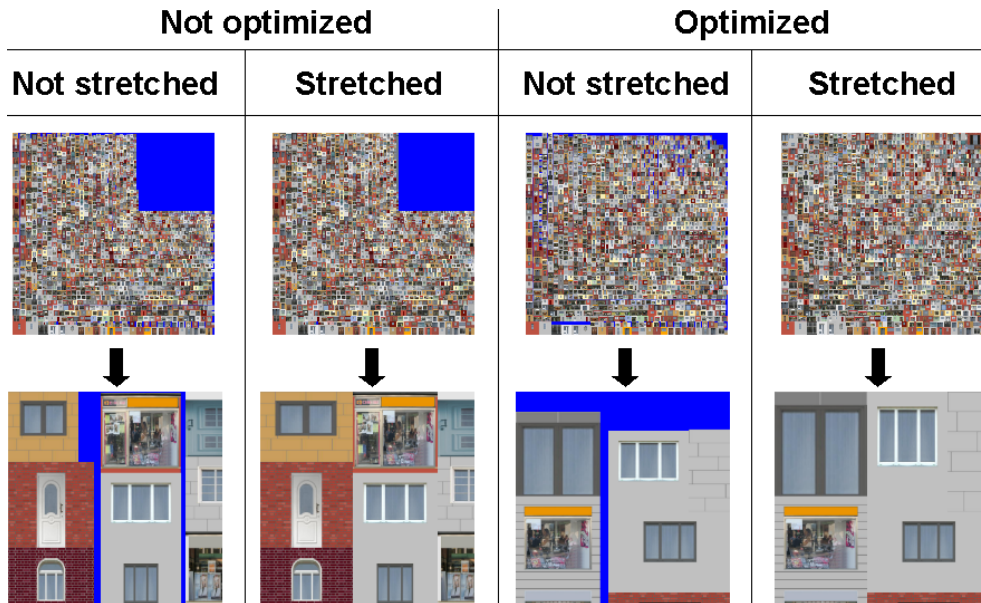
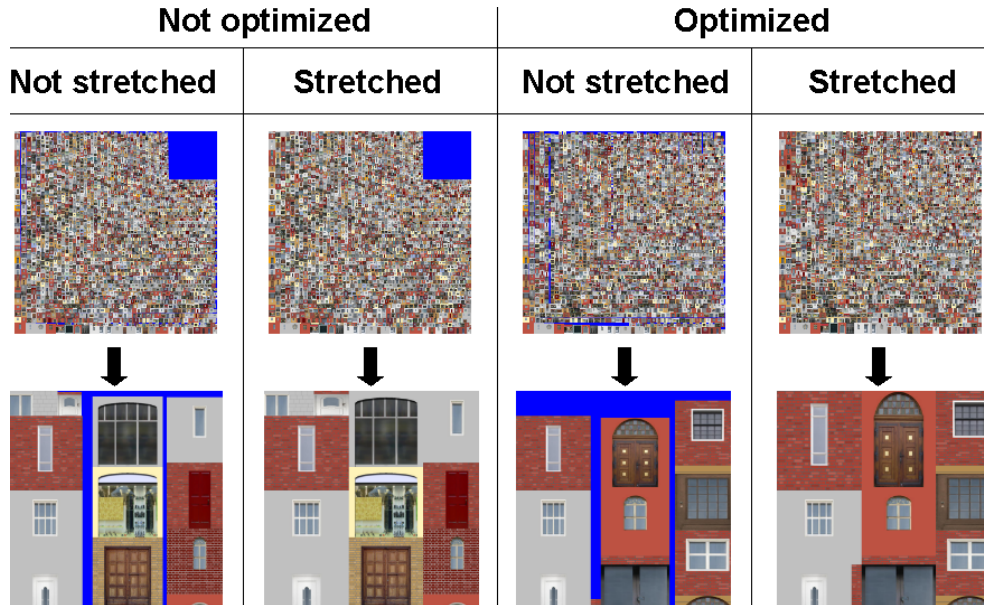


Figure 55: Texture set 5 packing



The Table 14 and Figure 56 shows for each texture set the occupancy of the packing of each of the four methods. We see that the best option is the optimized and stretched, obtaining always a 100% coverage of the atlas.

Figure 56: Texture packing results

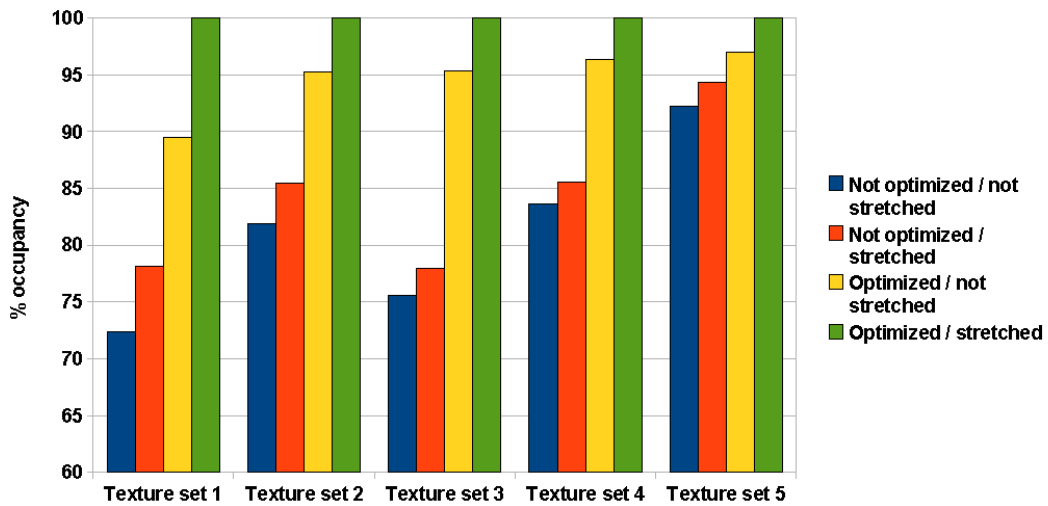


Table 14: Texture set packing occupancy

Texture set	1	2	3	4	5
Resolution	2048x2048	2048x2048	4096x4096	4096x4096	4096x4096
Number of subtextures	192	378	1107	1689	2897
meters/texel	0.02	0.03	0.03	0.04	0.05
Not optimized / not stretched	72.37%	81.85%	75.56%	83.65%	92.20%
Not optimized / stretched	78.14%	85.44%	77.96%	85.58%	94.32%
Optimized / not stretched	89.48%	95.29%	95.34%	96.36%	96.96%
Optimized / stretched	100%	100%	100%	100%	100%

10.3 Encoding texture chart coordinates

We have tested the three chart encoding options listed in Table 1 of Section 8.1.1. We rendered the whole test model (see Section 9.1) to extract the results shown on the Figure 15. The vertices per second are measured using a viewport of 1×1 with the camera facing the whole geometrical dense city. The frames per seconds are measured with the maximum size of the viewport (in our case 1440×900) and facing the whole city. Finally, the fragments per second are measured rendering one building and facing a textured facade with a maximum size of the viewport. Since Option 1 in Table 1 does not have VBO support we adapted the rendering scheme to use VBO to make it comparable with the other two options. We introduced the additional condition to have the same chart per vertex buffer. This increases the number of required VBO buffers and the fragmentation reducing the performance.

The results are better than we predicted. The option that uses packed coordinates (Option 3) have the highest vertices/s and frames/s rate although it have to decompress each vertex information (see Section 8.1.3). The Option 2 do not have any decompression step but has 20.76% and 23.67% less vertices/s and frames/s rate respectively. The option 1 fails with both intensive geometry tests but has the best fragments/s rate because it has the simplest fragment shader. So we see that the option 3 has more performance reducing the memory bandwidth instead it have an additional decompression cost per vertex.

Table 15: Chart encoding performance (for more information of the encoding techniques see Table 1)

Encoding technique	Vertices/s	Fragments/s	Frames/s
Option 1	44,032,002	1,570,752,000	14.57
Option 2	436,739,971	1,382,832,000	132.37
Option 3	527,426,950	1,362,096,000	163.71

Figure 57: Chart encoding performance (vertices/s)

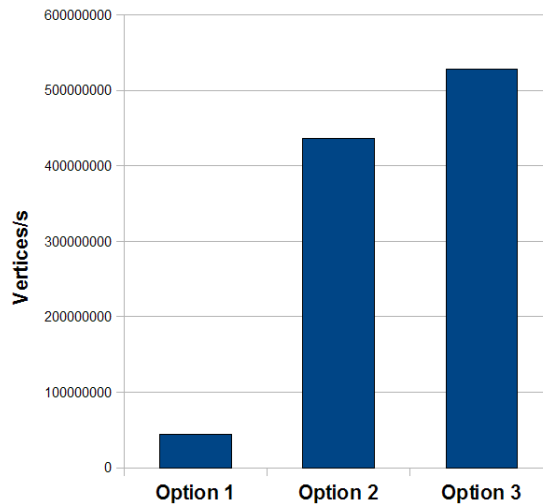


Figure 58: Chart encoding performance (fragments/s)

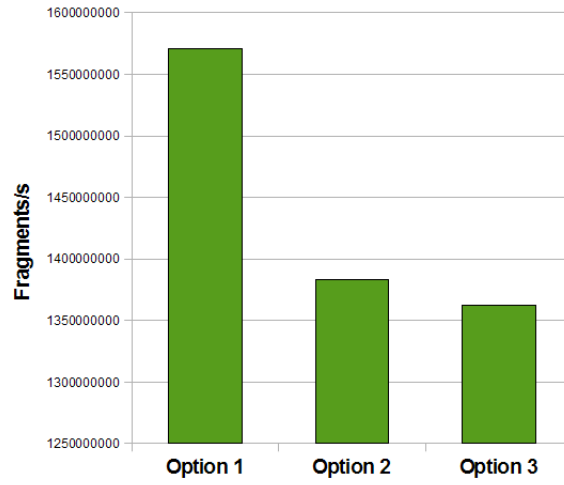
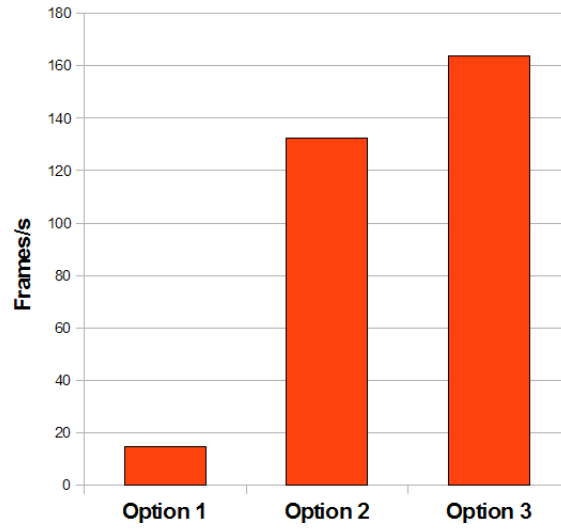


Figure 59: Chart encoding performance (frames/s)



11 Time Performance

To evaluate the time performance we have defined a walkthrough of the city going from far to a close view of a building. We show the results for each rendering technique. When we do not use texture atlas, we tried to load the original textures but it was just impossible to load all of them so we switched to a subsampled version with 256x256 and DXT1 compression for the results provided in this section. We do not use visual improvements (see Section VII) and disable terrain rendering to just evaluate the texture atlas rendering.

The Figure 61 shows the evolution of the framerate during the walk. The snapshots show the evolution of the walkthrough in steps of two seconds.

Table 16: Resulting framerate for each technique (walkthrough)

Technique	Minimum FPS	Maximum FPS	Average FPS
Encoding option 1	20	30.36	24.63
Encoding option 2	136	210.16	174.55
Encoding option 3	158	251.5	206.89
VBO / No texture atlas	10.3	14.22	12.34
Immediate / No texture atlas	1.16	1.59	1.16

Figure 60: Resulting framerate for each technique (walkthrough)

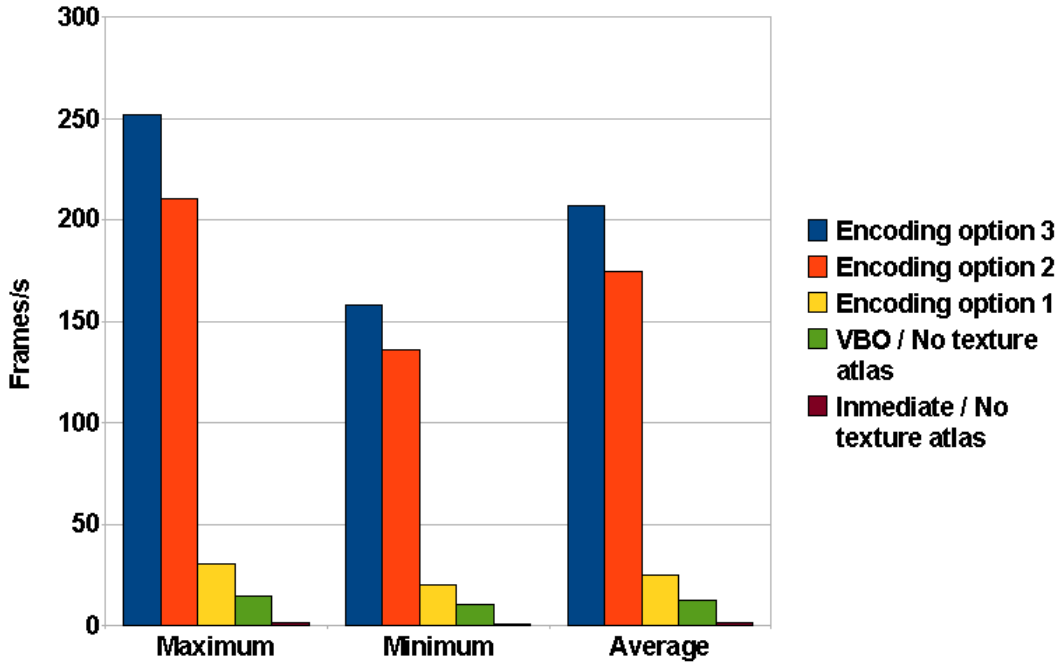
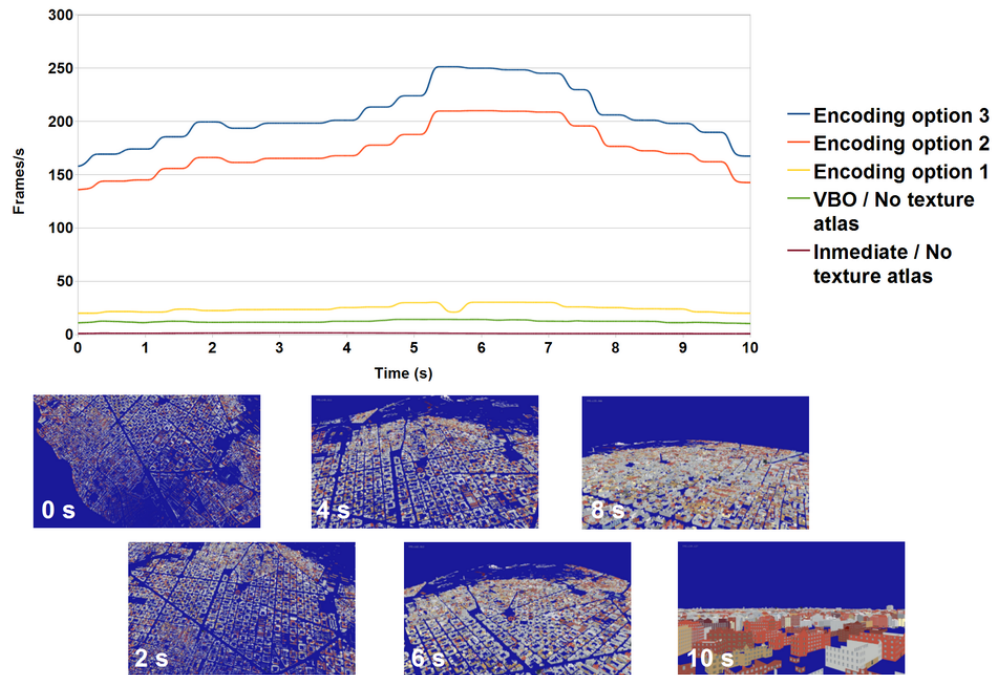


Figure 61: Framerate evolution of walkthrough



As in Section 10.3, we see that the encoding option 3 has the best framerate, followed by the encoding option 2. Both have similar evolution during the walkthrough. However, the encoding option 1 fails in the test (obtains FPS nearly the other two options that do not use texture atlas) because of the heavy VBO fragmentation. Comparing the average FPS of the encoding option 2 and the technique that use VBO without texture atlas, we have a speed-up factor of 17, very significant in a real-time rendering system.

12 Selected snapshots

The next figures are selected snapshots of the city using the visual improvements described at Section VII and enabling terrain rendering.

Figure 62: Barcelona snapshot 1



Figure 63: Barcelona snapshot 2



Figure 64: Barcelona snapshot 3



Figure 65: Barcelona snapshot 4

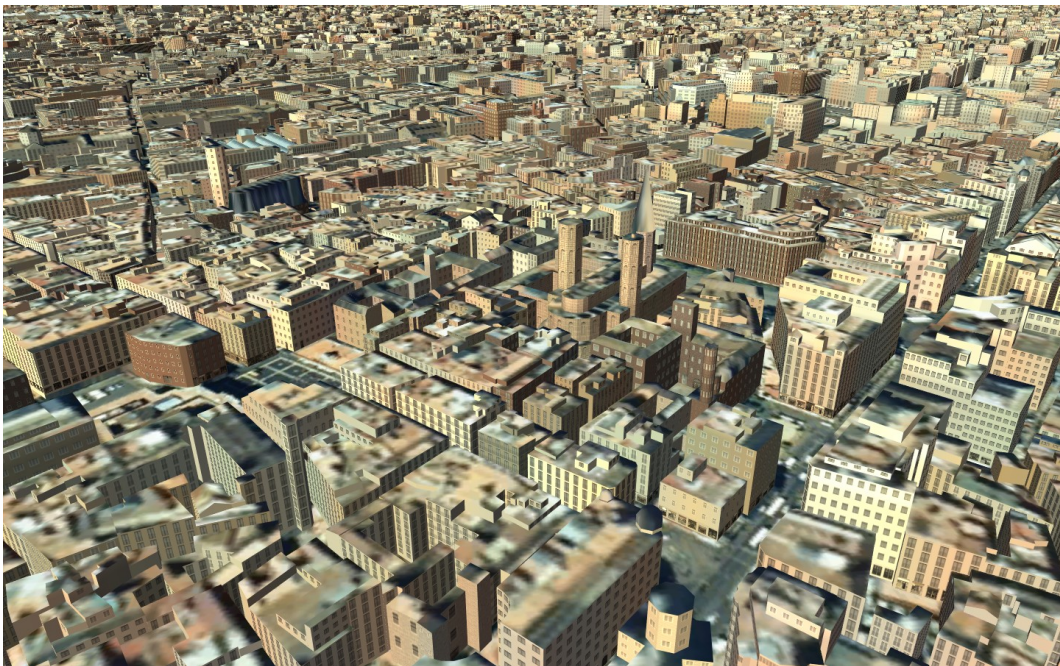


Figure 66: Barcelona snapshot 5



Figure 67: Barcelona snapshot 6



Part VI

Conclusions

We have developed a novel technique that generates space-optimized texture atlas encoding repetitive texture details of the geometry and implemented an application able to render in real time a huge city with thousands of textures achieving interactive framerates. Our contributions are summarized as follows:

- *Creating optimized texture atlas*
 - An algorithm for resizing each chart in accordance with the object-space size of the surface the chart is mapped onto and the perceptual importance under a given viewing conditions.
 - An algorithm to pack rectangular charts into a single texture that minimizes the unused space.
- *Rendering optimized texture atlas*
 - A compressed texture coordinate format designed to support tiled textures avoiding the unfolding of periodic textures. Several shader techniques providing within-chart tiling support and decompression of texture coordinates.
 - Full support to DXTC formats avoiding artifacts due the texture atlas compression.
 - A texture atlas hierarchy supporting implicit mipmap levels per atlas and providing explicit user-defined texture LOD.
 - A visual improvement in the particular case of the city of Barcelona mixing with the facade details the extracted characteristic colours.

Part VII

Appendix

Texture wrapping GLSL code

Algorithm 7 Texture wrapping vertex program (see Section 8.1.1)

```
#extension GL_EXT_gpu_shader4 : enable
// bit mask
// bits 31..22 (10 bits): 0/4;
// all subimages start at multiple-of-four positions
// bits 21..15 ( 7 bits): S/4;
// all subimages have multiple-of-four dimensions
// bits 14.. 9 ( 6 bits): integer part of original tex coordinate
// (max repeat factor: 63)
// bits 8.. 0 ( 9 bits): fract part of original tex coordinate
//(max subimage size: 512)
const uint exp2_22 = 4194304u;
const uint exp2_22_div4 = 1048576u;
const uint exp2_15 = 32768u;
const uint exp2_15_div4 = 8192u;
const uint exp2_9 = 512u;
varying vec4 rc; //input compressed coordinates
uniform vec2 textureSize; //texture atlas size
uniform int border; //texture atlas charts border size
void main(){

    // Decode input packed coordinates
    uvec2 rem;
    rc.st = vec2((input/exp2_22*4u)+border)/textureSize;
    rem = input % exp2_22;
    rc.pq = vec2((rem/exp2_15*4u)-2u*border)/textureSize;
    rem = rem % exp2_15;
    gl_TexCoord[0].st=vec2(rem/exp2_9)+vec2(rem%exp2_9)/vec2(exp2_9);
    gl_Position = ftransform();

}
```

Algorithm 8 Texture wrapping fragment program (see Section 8.1.3)

```
#extension GL_ARB_shader_texture_lod : enable
varying vec4 rc; //The decompressed texture coordinates
uniform sampler2D tex; //The input texture atlas
void main(){
    gl_FragColor=texture2DGrad(tex,rcv.pq*fract(gl_TexCoord[0].st)
        +rcv.st,dFdx(gl_TexCoord[0].st),dFdy(gl_TexCoord[0].st));
}
```

Quadtree generation

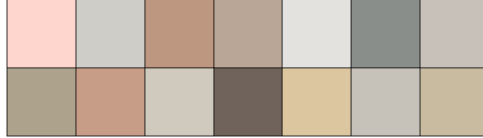
Algorithm 9 Quadtree generation (see Section 8.2.1)

```
//Quadtree constructor
//Each variable beginning with the name quadtree, refers to variables
//of the quadtree object
constructor Quadtree( $B, n$ )
if  $n = 0$  then
    // Base case
    quadtree. $B = B$ 
else
    // Recursive case
    for  $h = 0..4$  do
        // For each son collect buildings in the center of the quadrant
        vector Building subset;
        for  $i = 0..B.size()$  do
             $c = 0$ 
            if  $B[i].center.x > B.center.x$  then
                 $c = c + 1$ 
            end if
            if  $B[i].center.z > B.center.z$  then
                 $c = c + 2$ 
            end if
            if  $c = h$  then
                subset.push( $B[i]$ );
            end if
        end for
        quadtree.children[ $h$ ] = new QuadreeNode(subset,  $n - 1$ )
    end for
end if
```

Visual improvements

We have noticed that rendering the Barcelona model using the provided textures is quite unrealistic: the textures contains several details but the colours used do not correspond in some cases to the characteristic colours of Barcelona facades. Thanks to Jordi Moyes and Carlos Andujar, the most present colours of the buildings have been extracted. In total we have twenty-four different colours, some of them represented in the Figure 68.

Figure 68: Some characteristic colours of Barcelona facades



We want to mix this colour with the facade texture to improve the visual realism. The mixing is done in the fragment program stage. We mathematically define a subdivision of the XZ coordinates (the Y represents de building height) that changes above a variable step parameter β (meters). Let C be the collection with n characteristic colours and p_{frag} the fragment position, we have that:

$$t = \frac{|p_{fragx}| |p_{fragz}|}{\beta}$$

$$colour_{mix} = C[[t] \bmod n] (t - [t]) + C[([t] + 1) \bmod n] (1 - t + [t])$$

Where $colour_{mix}$ is the colour used for the mixing with the texture colour.

For the shading we only consider local diffuse and ambient shading and try to place the light position in a similar way that seems to be in the aerial photos.

References

- [1] Improve batching using texture atlases. Technical report, Nvidia Corporation, 2004.
- [2] Carlos Andujar and Jonas Martinez. Locally-adaptive texture compression. *Congreso Español de Informática Gráfica*, 2009.
- [3] J.E. Beasley. An exact two-dimensional non-guillotine cutting tree search procedure. *Operations Research*, 33:49–64, 1985.
- [4] Andrew C. Beers, Maneesh Agrawala, and Navin Chaddha. Rendering from compressed textures. In *Proceedings of SIGGRAPH 96*, Computer Graphics Proceedings, Annual Conference Series, pages 373–378, August 1996.
- [5] David Benson and Joel Davis. Octree textures. *ACM Trans. Graph.*, 21(3):785–790, 2002.
- [6] Henrik Buchholz and Jürgen Döllner. View-dependent rendering of multiresolution texture-atlases. In *Proceedings of the IEEE Visualization 2005*, pages 215–222, 2005.
- [7] Peter J. Burt and Edward H. Adelson. The laplacian pyramid as a compact image code. pages 671–679, 1987.
- [8] Graham Campbell, Thomas A. DeFanti, Jeff Frederiksen, Stephen A. Joyce, and Lawrence A. Leske. Two bit/pixel full color encoding. *SIGGRAPH Comput. Graph.*, 20(4):215–223, 1986.
- [9] Nathan A. Carr and John C. Hart. Meshed atlases for real-time procedural solid texturing. *ACM Trans. Graph.*, 21(2):106–131, 2002.
- [10] Glenn Chan. Towards better chroma subsampling. *SMPTE journal*, 2008.
- [11] K Chiu, M Herf, P Shirley, S Swamy, C Wang, and K Zimmerman. Spatially nonuniform scaling functions for high contrast images. In *Graphics Interface*, pages 245–253, 1993.
- [12] Paolo Cignoni, Fabio Ganovelli, Enrico Gobbetti, Fabio Marton, Federico Ponchio, and Roberto Scopigno. Planet-sized batched dynamic adaptive meshes (p-bdam). In *VIS '03: Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, page 20, Washington, DC, USA, 2003. IEEE Computer Society.
- [13] David Cline and Parris K. Egbert. Interactive display of very large textures. In *VIS '98: Proceedings of the conference on Visualization '98*, pages 343–350, Los Alamitos, CA, USA, 1998. IEEE Computer Society Press.
- [14] Michael F. Cohen, Jonathan Shade, Stefan Hiller, and Oliver Deussen. Wang tiles for image and texture generation. *ACM Trans. Graph.*, 22(3):287–294, 2003.
- [15] Daniel Cohen-Or and Yishay Levanoni. Temporal continuity of levels of detail in delaunay triangulated terrain. In *VIS '96: Proceedings of the 7th conference on Visualization '96*, pages 37–42, Los Alamitos, CA, USA, 1996. IEEE Computer Society Press.
- [16] Scott Daly. The visible differences predictor: an algorithm for the assessment of image fidelity. pages 179–206, 1993.
- [17] Leila De Floriani, Paola Magillo, and Enrico Puppo. Building and traversing a surface at variable resolution. In *VIS '97: Proceedings of the 8th conference on Visualization '97*, pages 103–ff., Los Alamitos, CA, USA, 1997. IEEE Computer Society Press.

- [18] David DeBry, Jonathan Gibbs, Devorah DeLeon Petty, and Nate Robins. Painting and rendering textures on unparameterized models. In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 763–768, New York, NY, USA, 2002. ACM.
- [19] K. Dowsland. Some experiments with simulated annealing techniques for packing problems. *European Journal of Operational Research*, 68:389–399, 1993.
- [20] J.K. Lenstra E. Aarts. *Local Search in Combinatorial Optimization*. Wiley, 1997.
- [21] M. R. Garey D. S. Johnson E. G. Coffman, Jr. and R. E. Tarjan. Performance bounds for level-oriented two-dimensional packing algorithms. *SIAM Journal on Computing*, 9:808–826, 1980.
- [22] Matthias Eck, Tony DeRose, Tom Duchamp, Hugues Hoppe, Michael Lounsbery, and Werner Stuetzle. Multiresolution analysis of arbitrary meshes. In *SIGGRAPH '95: Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, pages 173–182, New York, NY, USA, 1995. ACM.
- [23] Carl Erikson, Dinesh Manocha, and William V. Baxter, III. Hlods for faster display of large static and dynamic environments. In *I3D '01: Proceedings of the 2001 symposium on Interactive 3D graphics*, pages 111–120, New York, NY, USA, 2001. ACM.
- [24] M. R. Garey F. R. K. Chung and D. S. Johnson. On packing two-dimensional bins. *SIAM. J. on Algebraic and Discrete Methods*, 3:66–76, 1982.
- [25] Sandor P. Fekete and Jörg Schepers. On more-dimensional packing i: Exact algorithms, 1997.
- [26] Sandor P. Fekete and Jörg Schepers. On more-dimensional packing i: Modeling, 1997.
- [27] Christian Frueh, Russell Sammon, and Avidesh Zakhor. Automated texture mapping of 3d city models with oblique aerial imagery. In *3DPVT '04: Proceedings of the 3D Data Processing, Visualization, and Transmission, 2nd International Symposium*, pages 396–403, Washington, DC, USA, 2004. IEEE Computer Society.
- [28] Thomas A. Funkhouser and Carlo H. Séquin. Adaptive display algorithm for interactive frame rates during visualization of complex virtual environments. In *SIGGRAPH '93: Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, pages 247–254, New York, NY, USA, 1993. ACM.
- [29] Stefan Maierhofer Gerd Hesina and Robert F. Tobler. Texture management for high-quality city walk-throughs. *Proceedings of CORP -International Symposium on Information and Communication Technologies in Urban and Spatial Planning*, 25:305–308, 2004.
- [30] Allen Gersho and Robert M. Gray. *Vector quantization and signal compression*. Kluwer Academic Publishers, Norwell, MA, USA, 1991.
- [31] Bernd Girod. What’s wrong with mean-squared error? pages 207–220, 1993.
- [32] Andrew S. Glassner. *Principles of Digital Image Synthesis*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1994.
- [33] Rafael C. Gonzalez and Richard E. Woods. *Digital Image Processing (3rd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2006.
- [34] David J. Heeger and James R. Bergen. Pyramid-based texture analysis/synthesis. In *SIGGRAPH '95: Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, pages 229–238, New York, NY, USA, 1995. ACM.

- [35] Lewis E. Hitchner and Michael W. McGreevy. Methods for user-based reduction of model complexity for virtual planetary exploration. *Proc. SPIE*, 1913:622–636, 1993.
- [36] Hugues Hoppe. Optimization of mesh locality for transparent vertex caching. *ACM SIGGRAPH*, pages 269–276, 1999.
- [37] Arnaud E. Jacquin. Image coding based on a fractal theory of iterated contractive image. *IEEE Transactions on Image Processing*, vol. 1, issue 1, pp. 18–30, 1:18–30, 1992.
- [38] Alexei A. Efros James Hays, Marius Leordeanu and Yanxi Liu. Discovering texture regularity as a higher-order correspondence problem. *ECCV*, 3952:522–535, 2006.
- [39] Nebojsa Jojic, Brendan J. Frey, and Anitha Kannan. Epitomic analysis of appearance and shape. In *ICCV '03: Proceedings of the Ninth IEEE International Conference on Computer Vision*, page 34, Washington, DC, USA, 2003. IEEE Computer Society.
- [40] A. J. Ahumada Jr. Simplified vision models for image quality assessment. *SID International Symposium Digest of Technical Papers*, 27:397–400, 1996.
- [41] G. Knittel, A. Schilling, A. Kugler, and W. Strasser. Hardware for superior texture performance. *Computers & Graphics*, 20:475–481, 1996.
- [42] Ali Lakhia. Efficient interactive rendering of detailed models with hierarchical levels of detail. In *3DPVT '04: Proceedings of the 3D Data Processing, Visualization, and Transmission, 2nd International Symposium*, pages 275–282, Washington, DC, USA, 2004. IEEE Computer Society.
- [43] Gregory Ward Larson, Holly Rushmeier, and Christine Piatko. A visibility matching tone reproduction operator for high dynamic range scenes. *IEEE Transactions on Visualization and Computer Graphics*, 3(4):291–306, 1997.
- [44] Sylvain Lefebvre. *Filtered Tilemaps (in Shader X6)*, chapter ., page (to appear). Shader X6. Charles River Media, 2008.
- [45] Sylvain Lefebvre and Carsten Dachsbacher. Tiletrees. In *I3D '07: Proceedings of the 2007 symposium on Interactive 3D graphics and games*, pages 25–31, New York, NY, USA, 2007. ACM.
- [46] Sylvain Lefebvre, Jerome Darbon, and Fabrice Neyret. Unified texture management for arbitrary meshes. Technical report, INRIA, 2004.
- [47] Aaron E. Lefohn, Shubhabrata Sengupta, Joe Kniss, Robert Strzodka, and John D. Owens. Glift: Generic, efficient, random-access gpu data structures. *ACM Trans. Graph.*, 25(1):60–99, 2006.
- [48] Joshua Levenberg. Fast view-dependent level-of-detail rendering using cached geometry. In *VIS '02: Proceedings of the conference on Visualization '02*, pages 259–266, Washington, DC, USA, 2002. IEEE Computer Society.
- [49] Bruno Lévy, Sylvain Petitjean, Nicolas Ray, and Jérôme Maillot. Least squares conformal maps for automatic texture atlas generation. *ACM Trans. Graph.*, 21(3):362–371, 2002.
- [50] M. Leyton. *Symmetry, Causality, Mind*. The MIT Press, 1992.
- [51] Wen-Chieh Lin, James H. Hays, Chenyu Wu, Vivek Kwatra, and Yanxi Liu. A comparison study of four texture synthesis algorithms on regular and near-regular textures. Technical Report CMU-RI-TR-04-01, Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, January 2004.
- [52] Peter Lindstrom, David Koller, William Ribarsky, Larry F. Hodges, Nick Faust, and Gregory A. Turner. Real-time, continuous level of detail rendering of height fields. In *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 109–118, New York, NY, USA, 1996. ACM.

- [53] Andrea Lodi, Silvano Martello, and Michele Monaci. Two-dimensional packing problems: A survey. *European Journal of Operational Research*, 141(2):241–252, September 2002.
- [54] Frank Losasso and Hugues Hoppe. Geometry clipmaps: terrain rendering using nested regular grids. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers*, pages 769–776, New York, NY, USA, 2004. ACM.
- [55] J. L. Mannos and D. J. Sakrison. The effects of a visual fidelity criterion on the encoding of images. *IEEE Transactions on Information Theory*, 20:522, 1974.
- [56] Silvano Martello, Michele Monaci, and Daniele Vigo. An exact approach to the strip-packing problem. *INFORMS J. on Computing*, 15(3):310–319, 2003.
- [57] W. Jr. Miller. *Symmetry Groups and Their Applications*. Academic Press: New York, 1972.
- [58] R.E. Gomory P.C. Gilmore. A linear programming approach to the cutting stock problem. *Operations Research*, 9:849–859, 1961.
- [59] Ken Perlin. An image synthesizer. In *SIGGRAPH '85: Proceedings of the 12th annual conference on Computer graphics and interactive techniques*, pages 287–296, New York, NY, USA, 1985. ACM.
- [60] Larry F. Hodges William Ribarsky Nick Faust Peter Lindstrom, David Koller and Gregory Turner. Level-of-detail management for real-time rendering of phototextured terrain. Technical report, Georgia Institute of Technology, 1995.
- [61] Paul Heckbert Pixar and Paul S. Heckbert. Survey of texture mapping. *IEEE Computer Graphics and Applications*, 6:56–67, 1986.
- [62] Budirijanto Purnomo, Jonathan D. Cohen, and Subodh Kumar. Seamless texture atlases. In *SGP '04: Proceedings of the 2004 Eurographics/ACM SIGGRAPH symposium on Geometry processing*, pages 65–74, New York, NY, USA, 2004. ACM.
- [63] Mahesh Ramasubramanian, Sumanta N. Pattanaik, and Donald P. Greenberg. A perceptually based physical error metric for realistic image synthesis. In *SIGGRAPH '99: Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, pages 73–82, New York, NY, USA, 1999. ACM Press/Addison-Wesley Publishing Co.
- [64] A. Ravishankar Rao and Gerald L. Lohse. Identifying high level features of texture perception. *CVGIP: Graph. Models Image Process.*, 55(3):218–233, 1993.
- [65] Holly E. Rushmeier, Gregory J. Ward, Christine D. Piatko, Phil Sanders, and Bert Rust. Comparing real and synthetic images: Some ideas about metrics. In *Rendering Techniques*, pages 82–91, 1995.
- [66] Marc Soucy, Guy Godin, and Marc Rioux. A texture-mapping approach for the compression of colored 3d triangulations. *The Visual Computer*, 12(10):503–514, 1996.
- [67] J.C Stevens and S.S Stevens. Brightness function: Effects of adaptation. *Journal of the Optical Society of America*, 53:375–385, 1963.
- [68] Christopher C. Tanner, Christopher J. Migdal, and Michael T. Jones. The clipmap: a virtual mipmap. In *SIGGRAPH '98: Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 151–158, New York, NY, USA, 1998. ACM.
- [69] Marco Tarini, Kai Hormann, Paolo Cignoni, and Claudio Montani. Polycube-maps. In *In Proceedings of SIGGRAPH 2004*, pages 853–860, 2004.

- [70] William C. Thibault and Bruce F. Naylor. Set operations on polyhedra using binary space partitioning trees. *SIGGRAPH Comput. Graph.*, 21(4):153–162, 1987.
- [71] Jack Tumblin and Holly Rushmeier. Tone reproduction for realistic images. *IEEE Computer Graphics and Applications*, 13(6):42–48, 1993.
- [72] Huamin Wang, Yonatan Wexler, Eyal Ofek, and Hugues Hoppe. Factoring repeated content within and among images. In *SIGGRAPH '08: ACM SIGGRAPH 2008 papers*, pages 1–10, New York, NY, USA, 2008. ACM.
- [73] Zhou Wang and Alan C. Bovik. *Mean Squared Error: Love It or Leave It?* IEEE Signal Processing Magazine, 1998.
- [74] Gregory J. Ward. The radiance lighting simulation and rendering system. In *SIGGRAPH '94: Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, pages 459–472, New York, NY, USA, 1994. ACM.
- [75] Li-Yi Wei. Tile-based texture mapping on graphics hardware. In *HWWS '04: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 55–63, New York, NY, USA, 2004. ACM.
- [76] Lance Williams. Pyramidal parametrics. *SIGGRAPH Comput. Graph.*, 17(3):1–11, 1983.
- [77] Wen-Chieh Lin Yanxi Liu and James H. Hays. Near-regular texture analysis and manipulation. *ACM SIGGRAPH*, 23, 2004.
- [78] Yangli Hector Yee and Anna Newman. A perceptual metric for production testing. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Sketches*, page 121, New York, NY, USA, 2004. ACM.
- [79] Eugene Zhang, Konstantin Mischaikow, and Greg Turk. Feature-based surface parameterization and texture mapping. *ACM Trans. Graph.*, 24(1):1–27, 2005.
- [80] Hualin Zhou, Min Chen, and Mike F. Webster. Comparative evaluation of visualization and experimental results using image comparison metrics. In *VIS '02: Proceedings of the conference on Visualization '02*, pages 315–322, Washington, DC, USA, 2002. IEEE Computer Society.
- [81] Jacob Ziv and Abraham Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, 24:530–536, 1978.